Consistency Checking in Complex Object Database Schemata with Integrity Constraints *

Domenico Beneventano°, Sonia Bergamaschi°, Stefano Lodi, Claudio Sartori

E-mail: {dbeneventano, sbergamaschi, slodi, csartori}@deis.unibo.it

Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna - CSITE-CNR
°Dipartimento di Scienze dell'Ingegneria, Università di Modena - CSITE-CNR

Abstract

Integrity constraints are rules which should guarantee the integrity of a database. Provided that an adequate mechanism to express them is available, the following question arises: is there any way to populate a database which satisfies the constraints supplied by a database designer? i.e., does the database schema, including constraints, admit at least a non–empty model?

This work gives an answer to the above question in a complex object database environment, providing a theoretical framework including the following ingredients: two alternative formalisms, able to express a relevant set of state integrity constraints with a declarative style; two specialized reasoners, based on the *tableaux calculus*, able to check the consistency of complex objects database schemata expressed with the two formalisms. The proposed formalisms share a common kernel, which supports complex objects and object identifiers, and allow the expression of acyclic descriptions of: classes, nested relations and views, built up by means of the recursive use of record, quantified set and object type constructors and by the intersection, union and complement operators. Furthermore, the kernel formalism allows the declarative formulation of typing constraints and integrity rules. In order to improve the expressiveness and maintain decidability of the reasoning activities we extend the kernel formalism in two alternative directions. The first formalism, $\mathcal{OLCP}$, introduces the capability of expressing path relations. As cyclic schemas are extremely useful, we introduce a second formalism, $\mathcal{OLCD}$, with the capability of expressing cyclic descriptions but disallowing the expression of path relations. In fact, we show that the reasoning activity in $\mathcal{OLCDP}$ (i.e. $\mathcal{OLCP}$ with cycles) is undecidable.

Index Terms: Database models, database semantics, consistency checking, complex object models, description logics, integrity constraints, object–oriented database, semantic integrity, subsumption.

1 Introduction and Motivation

Integrity constraints are rules which should guarantee the integrity of a database. Most database systems include support for integrity constraints: in RDBMSs and OODBMSs some typing constraints can be directly represented at schema definition time using the DDL; others are expressed and enforced by mechanisms such as *check conditions* and *assertions* in RDBMSs or specific *methods* in OODBMSs.

However, as pointed out in [32], little attention has been paid to the problem of determining whether a set of constraints (or/and rules) is *consistent*, i.e., non-contradictory or satisfiable. Contributions to these topics are based on the capability of expressing constraints with a declarative style and come from different research areas: deductive databases, description logics languages and object–oriented databases.

Deductive databases offer a uniform solution to the satisfiability problem for the wide class of integrity constraints which can be expressed as *deduction rules*, through the availability of general theorem provers, which

*Contact Address: Claudio Sartori, DEIS, Viale Risorgimento 2, I-40136 Bologna, Italy, Phone: +39 (51) 644.3554-(operator: 3548-3001), Fax: +39 (51) 644.3540, E-mail: csartori@deis.unibo.it

can be used both to check the consistency of a set of data definitions and constraints and to check database states with respect to constraints. Significant contributions on these topics are found in [21, 38, 41, 42].

A major drawback of these proposals is that they operate on a logical, flat data model, whereas actual directions of knowledge representations focus on powerful data modelling capabilities adding structural complexity to value-based relational databases. In fact, formalisms supporting classes with complex description organized in multiple inheritance hierarchies are a milestone both for database systems and knowledge representation systems developed in Artificial Intelligence [1, 4, 9, 17, 19, 27, 7, 31].

Among object-oriented database models [18, 25, 49], the class of the ones known as Complex Object Data Models (CODMs), recently considered in the area of Deductive Databases [2, 3], OODB [7], is particularly significant for the expressiveness of the data models and their precise extensional semantics: they support complex objects and object identifiers and allow the definition of complex values built by means of the recursive use of the set and the record constructors.

Furthermore, they provide techniques to guarantee inheritance consistency of database schemata, based on strict inheritance class taxonomies [1, 4, 7]. In the O_2 object-oriented DBMS [7], for example, the introduction of new classes which violate strict inheritance semantics is prevented by means of well-known type checking techniques [24] (i.e., a class named C can only be a specialization of a class named C' *if* the structural description of C is a *refinement* of the structural description of C').

From the structural point of view, also the Object–Relational database systems [49] adopt the same data model, with values and oids, but without a precise extensional semantics.

Among knowledge representation formalisms developed in Artificial Intelligence, the class of the ones known as description logics languages (*DLs*) [48], derived from the KL-ONE model [20], is particularly relevant for its object–oriented data modelling capabilities and for its techniques to check consistency of strict multiple inheritance taxonomies of classes.

DLs are a subset of first order logic: they express *concepts* (roughly classes) as logical formulas which contain one free variable (to be filled with instances of a concept) and are built using unary and binary predicates (connected by intersection, union and complement). *DLs* differentiate *primitive* and *defined* concepts, representing, respectively, only necessary conditions and necessary and sufficient conditions for objects to be instances of a concept. Thus, primitive concepts correspond to the usual semantics of database classes, whereas defined concepts correspond to the semantics of views. Given an extensional semantics for concepts and exploiting the semantics for defined concepts, DLs provide reasoning techniques, called *taxonomic reasoning*: *subsumption* computation (i.e. computation of *isa* relations not explicitly given but implied by the descriptions of concepts) and *inconsistency* detection (i.e. detection of always empty concepts). The complexity of these inferences was assessed for a variety of acyclic DLs (see e.g., [28]).

Recently, description logic–based formalisms for databases, which extend in different directions the above cited DLs and exploit taxonomic reasoning in a database environment, have been proposed [10, 12, 16, 17, 22, 23]. The extensions proposed in [16] were necessary to represent the richer expressiveness of CODMs with respect to previously proposed DLs. In fact, *CODMs* are, in some aspects, more expressive. First of all, CODMs introduce a distinction between *values* and *objects* with identity and, thus, between *value types* and *class types* (which are also briefly called *classes*), whereas DLs support only objects and classes. Further, CODMs support additional type constructors, such as *record* and *set* allowing complex values to be built. Finally, CODMs usually support the representation and management of *cyclic classes*, i.e., classes which directly or indirectly make references to themselves.

The capability of supporting value-types firstly introduced in the extended description logics, *ODL*, proposed in [15, 16] on the one hand permits a data model which is a compatible extension of the relational model and, on the other hand, it is fundamental to represent the so-called integrity *typing constraints*, which are described by predicates on concrete domains (integer, string, real) [1].

In this paper we propose a solution to the consistency problem in complex object database environment, providing a theoretical framework including: two alternative formalisms, $\mathcal{OLCP}$ and $\mathcal{OLCD}$, able to express a

[1] The relevance of typing constraints to build up concept description was recognized and dealt with in the DLs community only by Franz Baader in [5].

relevant set of state integrity constraints in a declarative style; two specialized reasoners, based on the *tableaux calculus*, able to check the consistency of complex objects database schema expressed by these formalisms.

$\mathcal{OLCP}$ and $\mathcal{OLCD}$ are not proposed as new data models, but as formal tools in which the structural part (i.e. disregarding the behavioral aspects) of the different existing OODB models [2, 3, 7, 39] can be mapped, and a class of integrity constrains can be expressed in a declarative style. Their precise extensional semantics, in the tradition of description logics, is a pre–requisite for the development of a specialized reasoner for the consistency checking.

The proposed formalisms share a common kernel, $\mathcal{OLC}$ (*Object Language with Complements*), which provides a general value-based and identity-based type system, disallowing cyclic descriptions. The descriptions of classes and nested relations come in two flavours: primitive and *virtual* (corresponding to defined semantics of DLs and views in DB environment). Type descriptions can be built by means of the recursive use of record and set type constructors (existentially and universally quantified) and by using: comparison operators; union, intersection, complement operators; *attribute paths*. Attribute paths are dot-separated sequences of attributes expressing navigation through classes and types of a complex object data schema. They are necessary for query languages in an OODBMS environment (see the examples in [36]) and were introduced to express constraints in [26].

$\mathcal{OLC}$ has been individuated as starting point, as it is a good compromise between expressiveness of the model and decidability of the reasoning activities. In fact, in $\mathcal{OLC}$ we can express significant state integrity constraints, i.e. *typing constraints* and *integrity rules*. A *typing constraint* expresses a comparison between an attribute path and a base value. The union, intersection and complement operators allow the expression of *if then integrity rules*. Furthermore, decidability results of the reasoning activities have been obtained in description logics research area for a comparable formalism, $\mathcal{ALC}$ [5, 35, 46].

There are many directions of possible extensions to the kernel formalism $\mathcal{OLC}$, which may be considered, but our main guideline is decidability. Thus we start from the formalism $\mathcal{OLCP}$ presented in section 2 and consider possible independent and relevant extensions: *path relations*, cyclic type names, paths across multi-valued attributes. The first, i.e. the $\mathcal{OLCP}$ (*Object Language with Complements and Path relations*) formalism, introduces the capability of expressing *path relations* (a *path relation* expresses a comparison between two attribute paths). The second, i.e. $\mathcal{OLCD}$ (*Object Language with Complements allowing Descriptive cycles*), introduces the capability of expressing cyclic descriptions. We prove that consistency checking in $\mathcal{OLCP}$ and $\mathcal{OLCD}$ is decidable. Unfortunately, we prove that for the formalism $\mathcal{OLCDP}$, allowing both the capability of expressing cyclic descriptions and the capability of expressing path relations, the reasoning activity becomes undecidable. Furthermore, extending $\mathcal{OLCP}$ with paths across multi-valued attributes leads to undecidability.

$\mathcal{OLCP}$ and $\mathcal{OLCD}$ are derived from the *ODL* formalism proposed in [16]: $\mathcal{OLCD}$ extends *ODL* with union, complement operators and existential quantification of sets; $\mathcal{OLCP}$ extends *ODL* in some features: union, complement operators and path relations but introduces the restriction of aciclicity of descriptions. Furthermore, consistency checking in *ODL* has been performed by means of completely different techniques.

The specialized reasoners we developed for $\mathcal{OLCP}$ and $\mathcal{OLCD}$ are based on the *tableaux calculus* [47] and extend the theoretical results obtained in [5, 35, 46]. These extensions are quite complex, since $\mathcal{OLCP}$ and $\mathcal{OLCD}$ represent combinations of features which are separately found in different proposals and, more of all, completely new techniques to detect the incoherence of path relations and typing constraints including value–types and objects have been developed.

The behavior of the reasoners is the following: given a schema and one of its class descriptions (eventually including integrity constraints), it returns true if, and only if, the class is *consistent* with respect to the schema, i.e., if at least a legal instance of the database schema exists such that the class interpretation is non–empty. The reasoner can also use consistency checking to perform subsumption computation, as consistency and subsumption can be reduced to each other.

The structure of the paper is the following. Subsection 1.1 illustrates our approach with some informal examples. In Section 2, the $\mathcal{OLCP}$ syntax and semantics are presented and consistency is formally defined.

Section 3 introduces the definitions necessary to develop the tableaux calculus of the specialized reasoners; gives the algorithm for checking consistency in $\mathcal{OLCP}$ schemata and the termination, soundness, completeness theorems. Furthermore, an application example of the consistency checking algorithm in $\mathcal{OLCP}$ is presented.

```
relation Level    =  [qualification: String, parameter: Integer]
class Employee    =  [name: String, salary: Integer]
class Manager     =  isa Employee and [level: Level]
class Repository  =  [ denomination: String or [name: String, address: String],
                       stock: {Material}]
class Department  =  [denomination: String, manager: Manager]
class Warehouse   =  isa Department and Repository
class Material    =  [name: String, risk: Integer]
class Fluid       =  isa Material and [viscosity: Integer]
class Solid       =  isa Material and not Fluid
class Shipment    =  [urgency: Integer, item: Material]
```

Table 1: The Company domain schema in OODB–like syntax

In Section 4, the syntactical and semantical differences of $\mathcal{OLCD}$ with respect to $\mathcal{OLCP}$ are described; the consistency checking algorithm in $\mathcal{OLCD}$ with the termination, soundness, completeness theorems are given. Furthermore, an application example of the consistency checking algorithm in $\mathcal{OLCD}$ is presented. Finally, the extension of $\mathcal{OLCP}$ with paths across multi-valued attributes is introduced and an undecidability result is given. In section 5, we discuss the related literature and the main integrity constraints which are not considered by our formalisms, and in section 6 we sketch the possible evolution of our work. The two appendices are concerned only with technical aspects: In Appendix A the proofs of the logical properties of the calculus, namely termination, soundness, completeness for $\mathcal{OLCP}$ and $\mathcal{OLCD}$, are given, and in Appendix B we show that for the formalism $\mathcal{OLCDP}$ allowing the expression of both cyclic descriptions and path relations, the reasoning activity becomes undecidable.

1.1 Examples

This subsection illustrates, with some examples, the types of integrity constraints which can be expressed with the proposed formalism and the contradictions that can be found.

Let us introduce some descriptions relative to a part of the organizational structure of a Company: "Employees have a name and earn a salary. Managers are employees and have a level composed of a qualification, which is a string, and a parameter, which is an integer. Repositories have a denomination, which can be either a string or a structure composed by a repository name and an address; a repository stocks a set of items which are all materials. Materials are described by a name and a risk. Departments have a denomination (string), and are managed by a manager. Warehouses have both the properties of departments and repositories. Fluids are materials and have a viscosity property. Solids are materials that are not in fluids. Shipments of materials are described by an urgency." Using OODB–like syntax, the Company domain example can be described as in Table 1.

The *relation* `Level` is a set of tuples, as in the relational model, whereas the classes are sets of identified objects. "Class" and "relation" are shorthands for primitive class and primitive relation, i.e. classes and relations whose extensions are assigned by the user.

The intersection operator (**and**) allows the expression of *multiple inheritance* (see class `Warehouse`). In particular, a subclass can redefine a field inherited from one (or more) of its its superclasses. In this case, due to our strict inheritance semantics, we compute the resulting type of the field as the intersection of the given types. For example, in the class `Manager` we could redefine the field `salary` as a subrange of `Integer`. On the other hand, if we redefine this field as a `String`, the class `Manager` is inconsistent. The complement operator (**not**) allows *exclusive* specialization hierarchies to be expressed (see classes `Fluid` and `Solid`). Furthermore, in class `Repository` the union operator (**or**) describes two alternative structures for the attribute `denomination` and the set operator {} introduces `stock` as a *multi-valued attribute*.

A notable capability of our formalism is attribute path support. For example, let us add to the schema the following class descriptions: "Dangerous materials (DangerMaterial) are materials with a risk greater than 5. Top managers (TopManager) are managers with a parameter greater than 10. Technicians are employees who work in a department; they earn less than the manager of the department where they work".

$$\begin{aligned}
\textbf{class}\ \texttt{DangerMaterial} &= \textbf{isa}\ \texttt{Material}\ \textbf{and}\ (\texttt{risk} > 5) \\
\textbf{class}\ \texttt{TopManager} &= \textbf{isa}\ \texttt{Manager}\ \textbf{and}\ (\texttt{level.parameter} > 10) \\
\textbf{class}\ \texttt{Technician} &= \textbf{isa}\ \texttt{Employee}\ \textbf{and}\ [\texttt{works}: \texttt{Department}] \\
&\quad \textbf{and}\ (\texttt{salary} < \texttt{works.manager.salary})
\end{aligned}$$

To express path undefinedness, we allow an **undef** operator. For instance, the following class VShipment describes shipments of materials which either do not have a viscosity property (such as solids) or whose viscosity is greater than 10:

$$\textbf{class}\ \texttt{VShipment} = \textbf{isa}\ \texttt{Shipment}\ \textbf{and}\ ((\textbf{undef}\ \texttt{item.viscosity})\ \textbf{or}\ (\texttt{item.viscosity} > 10))$$

The logical operators **and**, **or** and **not** allow the expression of relevant integrity constraints, that is, *if then* integrity rules. For example, let us specify for the class Shipment the following integrity constraint: "**for all** shipments it must hold that **if** (item.risk > 3), i.e., the risk of the material is greater than 3, **then** (urgency > 10), i.e., its urgency must be greater than 10". By translating implication, we can integrate the constraint in the class description, obtaining the following description for Shipment:

$$\begin{aligned}
\textbf{class}\ \texttt{Shipment} &= [\texttt{urgency}: \texttt{Integer}, \texttt{item}: \texttt{Material}] \\
&\quad \textbf{and}\ (\textbf{not}(\texttt{item.risk} > 3)\ \textbf{or}\ (\texttt{urgency} > 10))
\end{aligned}$$

The above examples show how class structures and integrity constraints can be represented in a unified framework, by embedding integrity constraints in class descriptions. Consequently, by assigning a precise semantics to the formalism constructors, this framework can be the basis to perform a global consistency checking.

Let us examine some examples of inconsistency. The following three classes show inconsistency due to type conflicts between inherited attributes and locally redefined attributes:

$$\begin{aligned}
\textbf{class}\ \texttt{Storage} &= \textbf{isa}\ \texttt{Warehouse}\ \textbf{and}\ [\texttt{denomination}: [\texttt{name}: \texttt{String}, \texttt{address}: \texttt{String}]] \\
\textbf{class}\ \texttt{AWarehouse} &= \textbf{isa}\ \texttt{Warehouse}\ \textbf{and}\ [\texttt{stock}: \{\texttt{Fluid}\}] \\
\textbf{class}\ \texttt{BWarehouse} &= \textbf{isa}\ \texttt{AWarehouse}\ \textbf{and}\ [\texttt{stock}: \{\textbf{exists}\ \texttt{Solid}\}]
\end{aligned}$$

Class Storage inherits from Warehouse a string attribute denomination and redefines it as a tuple. Class AWarehouse stocks a set of objects which belong all to the class Fluid. Class BWarehouse stocks at least a Solid (this is expressed by the existentially quantified set operator **exists**), but this is in conflict with the definition of its superclasses AWarehouse, which requires Fluid only in its stock (remember that Fluid and Solid are disjoint).

The following examples show how less intuitive contradictions can occur:

$$\begin{aligned}
\textbf{class}\ \texttt{BossTechnician} &= \textbf{isa}\ \texttt{Technician}\ \textbf{and}\ (\textbf{self} = \texttt{works.manager}) \\
\textbf{class}\ \texttt{SlowDangerShipment} &= \textbf{isa}\ \texttt{Shipment}\ \textbf{and}\ [\texttt{item}: \texttt{DangerMaterial}] \\
&\quad \textbf{and}\ (\texttt{urgency} < 5)
\end{aligned}$$

The class BossTechnician exploits paths and the *self* reference, and describes technicians who work in the departments they manage. Note that, due to the equality predicate (**self** = works.manager), we have that each object of the class BossTechnician must be an object of the class Manager. Furthermore, on the basis of the constraint expressed on the superclasses Technician, we obtain that a BossTechnician earn less than himself. On the other hand, if we modify the BossTechnician description as follows:

$$\textbf{class}\ \texttt{BossHomonTechnician} = \textbf{isa}\ \texttt{Technician}\ \textbf{and}\ (\textbf{self}.\texttt{name} = \texttt{works.manager.name})$$

we describe technicians who work in the departments managed by manager with the same name and no contradiction occurs.

Class SlowDangerShipment is inconsistent since DangerMaterial is constrained to a risk value greater than 5 and therefore its shipment requires an urgency value greater than 10.

As an example of relation inconsistency, let us consider the following definitions:

$$\underline{\textbf{relation}}\ \texttt{Skill} \quad = \quad \texttt{String}\ \underline{\textbf{and}}\ ((\underline{\textbf{self}} = \textit{"Typist"})\ \underline{\textbf{or}}\ (\underline{\textbf{self}} = \textit{"Secretary"}))$$

$$\underline{\textbf{relation}}\ \texttt{Requests} \quad = \quad [\texttt{skill}_1 : \texttt{Skill}, \texttt{skill}_2 : \texttt{Skill}, \texttt{skill}_3 : \texttt{Skill}]$$
$$\underline{\textbf{and}}\ (\texttt{skill}_1 \neq \texttt{skill}_2)\ \underline{\textbf{and}}\ (\texttt{skill}_2 \neq \texttt{skill}_3)\ \underline{\textbf{and}}\ (\texttt{skill}_3 \neq \texttt{skill}_1)$$

No tuple is allowed for `Requests`, since it requires three different values of `Skill`, while two only are available. Note that if `Skill` were a class, instead of a relation, we could instantiate it with many object associated to the value *"Typist"* or *"Secretary"* and then `Requests` would be consistent.

We can express a set of database *views* by introducing the semantics of *virtual types*, that is type names whose descriptions represent a set of necessary and sufficient conditions and we can check their consistency. For example:

$$\underline{\textbf{virtual class}}\ \texttt{SuperFluid} \quad = \quad \underline{\textbf{isa}}\ \texttt{Fluid}\ \underline{\textbf{and}}\ (\texttt{viscosity} < 10)$$
$$\underline{\textbf{virtual class}}\ \texttt{TopFluid} \quad = \quad \underline{\textbf{isa}}\ \texttt{Fluid}\ \underline{\textbf{and not}}\ \texttt{SuperFluid}\ \underline{\textbf{and}}\ (\texttt{viscosity} < 3)$$

the virtual class `TopFluid` is detected as inconsistent; note that if `SuperFluid` is a primitive class, instead of a virtual class, no inconsistency would be detected but an exclusive specialization hierarchy would be expressed.

Let us consider now some examples of the computation of *subsumption* relationships. Let us consider the following subschema, where the constraint on `Technician`'s salary has been removed (in this way, the class `BossTechnician` is now consistent) and the class `BossHomonTechnician` is now a virtual class:

$$\underline{\textbf{class}}\ \texttt{Department} \quad = \quad [\texttt{denomination} : \texttt{String}, \texttt{manager} : \texttt{Manager}]$$
$$\underline{\textbf{class}}\ \texttt{Technician} \quad = \quad \underline{\textbf{isa}}\ \texttt{Employee}\ \underline{\textbf{and}}\ [\texttt{works} : \texttt{Department}]$$
$$\underline{\textbf{class}}\ \texttt{BossTechnician} \quad = \quad \underline{\textbf{isa}}\ \texttt{Technician}\ \underline{\textbf{and}}\ (\underline{\textbf{self}} = \texttt{works.manager})$$
$$\underline{\textbf{virtual class}}\ \texttt{BossHomonTechnician} \quad = \quad \underline{\textbf{isa}}\ \texttt{Technician}\ \underline{\textbf{and}}\ (\underline{\textbf{self}}\texttt{.name} = \texttt{works.manager.name})$$

`BossTechnician` is subsumed by `BossHomonTechnician`, since if one is the manager of the department for which he works, of course he has the same name. It is important to observe that this subsumption holds only if `BossHomonTechnician` is a virtual class. On the other hand, the semantics of virtual class isn't the only way to infer subsumptions: `BossTechnician` is subsumed by `Manager`, no matter if they are both primitive classes, since he manages his department, but we know that a department must be managed by an element of class `Manager`.

As a final example, let us consider the usage of virtual relations. A virtual relation collects all the values, at every nesting level, which are subsumed by its description. For instance, the collection of the tuples having a string attribute `name` is given as follows:

$$\underline{\textbf{virtual relation}}\ \texttt{Names} = [\texttt{name} : \texttt{String}]$$

The relation `Names` will include all tuples which are associated to objects from `Employee` and `Material` and the denominations of `Repository` which are not strings.

2 A Formalism for Complex Objects including Integrity Constraints

$\mathcal{OLCP}$ is an extension of the *object description language* ODL, introduced in [16] and is in the tradition of complex object data models [2, 39]. $\mathcal{OLCP}$, as its ancestor ODL, provides a *system of base types*: string, boolean, integer, real; the type constructors *tuple*, *set* and *class* allow the construction of complex value types and class types. Class types (also briefly called *classes*) denote sets of *objects with an identity and a value*, while *value types* denote sets of *complex, finitely nested values without object identity*. Additionally, an intersection operator can be used to create intersections of previously introduced types allowing simple and multiple inheritance. Finally, types can be given names. Named types come in two flavors: a named type may be *primitive*, which means that the user has to specify the membership of an *element* in the interpretation of the name or *virtual*, in which case its interpretation is computed.

$\mathcal{OLCP}$ extends ODL with the introduction of the union operator ($\sqcup$), of the complement operator ($\neg$) and of the existential set specification $\{S\}_\exists$. Another relevant extension is the introduction in the type system of the *path specifications* and *comparison operators* for the description of *predicates*. The main limitation with respect to ODL is that circular references are not allowed in type name descriptions, in order to guarantee the decidability of consistency checking. Moreover the type used in a set constructor cannot be another set type.

$$\delta: \begin{cases} o_1 & \mapsto & [\texttt{name}: \textit{“Liz”}, \texttt{salary}: \textit{50000}, \texttt{works}: o_2] \\ o_2 & \mapsto & [\texttt{denomination}: \textit{“Administration”}, \texttt{manager}: o_3] \\ o_3 & \mapsto & [\texttt{name}: \textit{“Rocco”}, \texttt{salary}: \textit{80000}, \texttt{level}: [\texttt{qualification}: \textit{“marketing”}, \texttt{parameter}: \textit{4}]] \\ o_4 & \mapsto & [\texttt{denomination}: \textit{“Metallurgy”}, \texttt{stock}: \{o_5, o_6\}] \\ o_5 & \mapsto & [\texttt{name}: \textit{“iron”}, \texttt{risk}: \textit{1}] \\ o_6 & \mapsto & [\texttt{name}: \textit{“steel”}, \texttt{risk}: \textit{2}] \\ o_7 & \mapsto & [\texttt{name}: \textit{“Ava”}, \texttt{salary}: \textit{90000}, \texttt{works}: o_2] \\ \vdots & & \end{cases}$$

Table 2: A possible assignment of values to objects

2.1 Values and Objects

We assume the union of the integers, the strings, the booleans, and the reals as the set $\mathcal{D}$ of *base values*. To build *complex values*, we further assume two disjoint countable sets of symbols $\mathbf{A}$ and $\mathcal{O}$, respectively called the *attributes* (denoted by $a, a_1, a_2, \ldots$), and the *object identifiers* (denoted by $o, o_1, o_2, \ldots$), both disjoint from $\mathcal{D}$. The set $\mathcal{V}$ of all *values over* $\mathcal{O}$ is defined as the smallest set containing $\mathcal{D}$ and $\mathcal{O}$, such that, if $v_1, \ldots, v_p$ are values, then the set $\{v_1, \ldots, v_p\}$ is a value, and a partial function $t\colon \mathbf{A} \to \{v_1, \ldots, v_p\}$ is a value. The function t is the usual tuple value; the standard notation $[a_1\colon v_1, \ldots, a_p\colon v_p]$ will be henceforth used.

Let $=, \neq, >, <, \geq, \leq$ be the equality, inequality and total order relations, denoted by θ, defined as usual on $\mathcal{D}$. Equality and inequality can be extended from $\mathcal{D}$ to all $\mathcal{V}$: the equality operator $(=)$ has the meaning of *identity*, i.e., two objects are equal if they have the same identifier, two sets are equal iff they have equal elements, two tuples, say $v_a = [a_1\colon v_1, \ldots, a_p\colon v_p]$ and $v_b = [a'_1\colon v'_1, \ldots, a'_q\colon v'_q]$, are equal if they have the same attributes and equal attribute labels are mapped to equal values.

Object identifiers are assigned values by a *total value function* δ from $\mathcal{O}$ to $\mathcal{V}$. For instance, we may have the assignment of values to objects of table 2.

2.2 Paths

A *path* p is either the symbol ϵ, or a dot-separated sequence of elements $e_1 . e_2 . \cdots . e_n$, where $e_i \in \mathbf{A} \cup \{\triangle\}$ $(i = 1, \ldots, n)$. ϵ denotes the unique path of length 0. Let $\mathbf{W}$ denote the set of all paths. Given a set of object identifiers $\mathcal{O}$ and a value function δ, we introduce the function $\mathcal{J}\colon \mathbf{W} \longrightarrow 2^{\mathcal{V}\times\mathcal{V}}$, mapping paths to partial functions on $\mathcal{V}$:

- empty path: $\mathcal{J}[\epsilon] = \Big\{(v, v) \in \mathcal{V} \times \mathcal{V}\Big\}$
- single element path: $\mathcal{J}[a] = \Big\{(v_1, v_2) \in \mathcal{V} \times \mathcal{V} \mid v_1 = [\ldots, a : v_2, \ldots]\Big\}$
  $\mathcal{J}[\triangle] = \Big\{(o, v) \in \mathcal{O} \times \mathcal{V} \mid \delta(o) = v\Big\}$
- multiple element path: $\mathcal{J}[e_1 . e_2 . \cdots . e_n] = \mathcal{J}[e_1] \circ \mathcal{J}[e_2] \circ \cdots \circ \mathcal{J}[e_n]$
  where $\circ$ is the symbol of function composition.

For instance, with the assignments of values to objects in table 2 we have:

$$\begin{aligned} \mathcal{J}[\triangle.\texttt{name}] &= \{(o_1, \textit{“Liz”}), (o_3, \textit{“Rocco”}), (o_5, \textit{“iron”}), (o_6, \textit{“steel”}), (o_7, \textit{“Ava”})\} \\ \mathcal{J}[\triangle.\texttt{stock}] &= \{(o_4, \{o_5, o_6\})\} \\ \mathcal{J}[\triangle.\texttt{level}] &= \{(o_3, [\texttt{qualification}: \textit{“marketing”}, \texttt{parameter}: 4])\} \\ \mathcal{J}[\texttt{parameter}] &= \{([\texttt{qualification}: \textit{“marketing”}, \texttt{parameter}: 4], 4)\} \\ \mathcal{J}[\triangle.\texttt{level.parameter}] &= \{(o_3, 4)\} \end{aligned}$$

Notice that, for all p, $\mathcal{J}[p]$ is undefined on set values. In section 4.3, paths defined also on set values will be discussed.

Let v be a value and p be a path. By $\mathcal{J}[p](v)$ we mean the unique value (when it exists) reachable from v following p, that is the value of the partial function $\mathcal{J}[p]$ in v.

2.3 Types and Schemas

We assume a countable set $\mathbf{N}$ of type names (denoted by N, N_1,N_2, ...), which includes the set $\mathbf{B} = \{$`Integer`, `String`, `Bool`, `Real`$\}$ of base-type designators (which will be denoted by B) and the symbols $\top$, $\bot$.

$\mathbf{S}(\mathbf{A}, \mathbf{N})$ denotes the set of all *finite type descriptions* (denoted by S, S_1, S_2, ...), also briefly called *types*, over given $\mathbf{A}, \mathbf{N}$, obtained according to the following abstract syntax rule, where $a_i \neq a_j$ for $i \neq j$ (in the sequel p, p_1, p_2, ... , denote a path, d denotes a base value):

$$\begin{aligned} S \quad \rightarrow \quad & N \mid S_1 \sqcup S_2 \mid S_1 \sqcap S_2 \mid \neg S \\ & \mid \{S\}_\forall \mid \{S\}_\exists \mid [a_1 : S_1, \ldots, a_k : S_k] \mid \triangle S \\ & \mid p_1 \theta p_2 \mid p \theta d \mid p\uparrow \end{aligned}$$

$\top$ denotes the *top type*, $\bot$ denotes the *empty type*, $\{\}_\forall$ and $[\ ]$ denote the usual type constructors of set and record (tuple), respectively. The $\{S\}_\exists$ construct is an existential set specification, where at least one element of the set must be of type S. The construct $\sqcap$ stands for *intersection*, the construct $\sqcup$ stands for *union*, the construct $\neg$ stands for *complement*, whereas $\triangle$ constructs class descriptions, i.e., is an object set forming constructor. $p_1 \theta p_2$, $p \theta d$, $p\uparrow$ represent *atomic predicates*: $p_1 \theta p_2$ is a *path relation*, $p \theta d$ is a *range restriction* and $p\uparrow$ expresses *path undefinedness*.

Given a set of type descriptions $\mathbf{S}(\mathbf{A}, \mathbf{N})$, a *schema* σ over $\mathbf{S}(\mathbf{A}, \mathbf{N})$ is a total function $\sigma\colon \mathbf{N} \setminus (\mathbf{B} \cup \{\top, \bot\}) \rightarrow \mathbf{S}(\mathbf{A}, \mathbf{N})$, which associates type names to descriptions. σ is partitioned into two functions: σ_P, which introduces the description of primitive type names whose extensions must be explicitly provided by the user; and σ_V, which introduces the description of virtual type names whose extensions can be recursively obtained from the extension of the types occurring in their description. Some of the types introduced in the examples of Section 1.1 are mapped into $\mathcal{OLCP}$ in Table 3.

Notice the intersection operator ($\sqcap$) constructs a type that satisfies all the constraints of its operand types. In particular, it can be used to express inheritance in a name description. Formally, N_1 *inherits directly from* N_2, iff $\sigma(N_1) = S_1 \sqcap \cdots \sqcap S_n$, $n > 0$, and $N_2 = S_i$ for some i, $1 \leq i \leq n$.

Observe that the $\triangle$ operator allows the distinction of object types from value types. *Class descriptions* are preceded by $\triangle$ (as in the case of `Material`), while descriptions without $\triangle$ are related to relation names (as in the case of `Level`). For instance, when we define a new class by adding properties to another class, the tuple of the new class must be preceded by $\triangle$, as for `Fluid`. On the other hand, OODB languages usually do not make this distinction. Notice also that the $\triangle$ symbol is used both as a path element and as a class constructor. The overloading matches the intuition that $\triangle$`.viscosity` < 10 is semantically equivalent to $\triangle$[`viscosity` : $(\epsilon < 10)$] (see the definition of interpretation function below).

In $\mathcal{OLCP}$ we impose two following syntactic limitations: 1) the nesting of set types is restricted, therefore a type such as $\{\{S\}_*\}_*$, where $*$ denotes both $\forall$ and $\exists$, is not allowed; 2) a $\mathcal{OLCP}$ schema is acyclic, therefore a type name may not refer directly or indirectly to itself. Formally, cycles are recognized by means of the notion of *dependence*: N_1 *depends on* N_2, where $N_1, N_2 \in \mathbf{N}$ if N_2 is contained in the expression defining N_1, $\sigma(N_1)$; a schema is *acyclic* iff the transitive closure of *depends on* is a strict partial order. Since an $\mathcal{OLCP}$ schema is acyclic, it is also *inheritance well-founded*, i.e., the transitive closure of the inheritance relation is a strict partial order.

2.4 Interpretations and Database Instances

In the following, we will write $\mathbf{S}$ instead of $\mathbf{S}(\mathbf{A}, \mathbf{N})$ when the components are obvious from the context.

$$
\begin{array}{rcl}
\sigma_P(\texttt{Level}) & = & [\texttt{qualification}: \texttt{String}, \texttt{parameter}: \texttt{Integer}] \\
\sigma_P(\texttt{Employee}) & = & \triangle[\texttt{name}: \texttt{String}, \texttt{salary}: \texttt{Integer}] \\
\sigma_P(\texttt{Manager}) & = & \texttt{Employee} \sqcap \triangle[\texttt{level}: \texttt{Level}] \\
\sigma_P(\texttt{Repository}) & = & \triangle[\ \texttt{denomination}: \texttt{String} \sqcup [\texttt{name}: \texttt{String}, \texttt{address}: \texttt{String}], \\
 & & \quad \texttt{stock}: \{\texttt{Material}\}_\forall] \\
\sigma_P(\texttt{Department}) & = & \triangle[\texttt{denomination}: \texttt{String}, \texttt{manager}: \texttt{Manager}] \\
\sigma_P(\texttt{Warehouse}) & = & \texttt{Department} \sqcap \texttt{Repository} \\
\sigma_P(\texttt{Material}) & = & \triangle[\texttt{name}: \texttt{String}, \texttt{risk}: \texttt{Integer}] \\
\sigma_P(\texttt{Fluid}) & = & \texttt{Material} \sqcap \triangle[\texttt{viscosity}: \texttt{Integer}] \\
\sigma_V(\texttt{SuperFluid}) & = & \texttt{Fluid} \sqcap (\triangle.\texttt{viscosity} < 10) \\
\sigma_P(\texttt{Solid}) & = & \texttt{Material} \sqcap \neg\texttt{Fluid} \\
\sigma_P(\texttt{Shipment}) & = & \triangle[\texttt{urgency}: \texttt{Integer}, \texttt{item}: \texttt{Material}] \\
 & & \sqcap(\neg(\triangle.\texttt{item}.\ \triangle\ .\texttt{risk} > 3)) \sqcup (\triangle.\texttt{urgency} > 10)) \\
\sigma_P(\texttt{TopManager}) & = & \texttt{Manager} \sqcap (\triangle.\texttt{level}.\texttt{parameter} > 10) \\
\sigma_P(\texttt{Technician}) & = & \texttt{Employee}\ \sqcap \triangle\,[\texttt{works}: \texttt{Department}] \\
 & & \qquad \sqcap\ (\triangle.\texttt{salary} < \triangle.\texttt{works}.\ \triangle\ .\texttt{manager}.\ \triangle\ .\texttt{salary}) \\
\sigma_P(\texttt{VShipment}) & = & \texttt{Shipment} \sqcap ((\triangle.\texttt{item}.\triangle.\texttt{viscosity} \uparrow) \sqcup (\triangle.\texttt{item}.\triangle.\texttt{viscosity} > 10)) \\
\sigma_P(\texttt{BossTechnician}) & = & \texttt{Technician} \sqcap\ (\epsilon = \triangle.\texttt{works}.\ \triangle\ .\texttt{manager}) \\
\sigma_P(\texttt{Skill}) & = & \texttt{String} \sqcap ((\epsilon = \textit{“Typist”}) \sqcup (\epsilon = \textit{“Secretary”})) \\
\sigma_P(\texttt{Requests}) & = & [\texttt{skill}_1: \texttt{Skill}, \texttt{skill}_2: \texttt{Skill}, \texttt{skill}_3: \texttt{Skill}] \\
 & & \sqcap(\texttt{skill}_1 \neq \texttt{skill}_2) \sqcap (\texttt{skill}_2 \neq \texttt{skill}_3) \sqcap (\texttt{skill}_3 \neq \texttt{skill}_1) \\
\sigma_P(\texttt{AWarehouse}) & = & \texttt{Warehouse} \sqcap \triangle[\texttt{stock}: \{\texttt{Fluid}\}_\forall] \\
\sigma_P(\texttt{BWarehouse}) & = & \texttt{AWarehouse} \sqcap \triangle[\texttt{stock}: \{\texttt{Solid}\}_\exists]
\end{array}
$$

Table 3: The company domain schema in $\mathcal{OLCP}$

Let $\mathcal{I}_{\mathbf{B}}$ be the (fixed) standard interpretation function from $\mathbf{B}$ to $2^{\mathcal{D}}$. For a given object assignment δ, each type expression S is mapped to a set of values (its interpretation). An *interpretation function* is a function $\mathcal{I}$ from $\mathbf{S}$ to $2^{\mathcal{V}}$ satisfying the following equations:

$$
\begin{aligned}
\mathcal{I}[\top] &= \mathcal{V} \\
\mathcal{I}[\bot] &= \emptyset \\
\mathcal{I}[B] &= \mathcal{I}_{\mathbf{B}}[B] \\
\mathcal{I}[\{S\}_\forall] &= \{M \mid M \subseteq \mathcal{I}[S]\} \\
\mathcal{I}[\{S\}_\exists] &= \{M \mid M \cap \mathcal{I}[S] \neq \emptyset\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{I}\Big[[a_1 : S_1, \ldots, a_p : S_p]\Big] &= \{t\colon \mathbf{A} \to \mathcal{V} \mid t(a_i) \in \mathcal{I}[S_i], 1 \leq i \leq p\} \\
\mathcal{I}[S_1 \sqcap S_2] &= \mathcal{I}[S_1] \cap \mathcal{I}[S_2] \\
\mathcal{I}[S_1 \sqcup S_2] &= \mathcal{I}[S_1] \cup \mathcal{I}[S_2] \\
\mathcal{I}[\neg S] &= \mathcal{V} \setminus \mathcal{I}[S] \\
\mathcal{I}[\triangle S] &= \Big\{o \in \mathcal{O} \Big| \delta(o) \in \mathcal{I}[S]\Big\} \\
\mathcal{I}[(p\theta d)] &= \Big\{v \in \mathcal{V} \mid \mathcal{J}[p](v)\theta d\Big\} \\
\mathcal{I}[(p_1\theta p_2)] &= \Big\{v \in \mathcal{V} \mid \mathcal{J}[p_1](v)\theta\mathcal{J}[p_2](v)\} \\
\mathcal{I}[(p\uparrow)] &= \{v \in \mathcal{V} \mid v \notin \text{ dom } \mathcal{J}[p]\}
\end{aligned}
$$

For instance, with the assignments of values to objects introduced in table 2 we have:

$$\begin{aligned}
\mathcal{I}\Big[[\texttt{stock}\colon \{(\triangle.\texttt{risk} \geq 2)\}_{\forall}]\Big] &= \emptyset \\
\mathcal{I}\Big[[\texttt{stock}\colon \{(\triangle.\texttt{risk} \geq 2)\}_{\exists}]\Big] &= \{[\texttt{denomination}\colon \textit{“Metallurgy”}, \texttt{stock}\colon \{o_5, o_6\}]\} \\
\mathcal{I}[(\triangle.\texttt{salary} < \triangle.\texttt{works}.\ \triangle.\texttt{manager}.\ \triangle.\texttt{salary})] &= \{o_1\} \\
\mathcal{I}[(\triangle\top) \sqcap (\triangle.\texttt{name}\uparrow)] &= \{o_2, o_4\}
\end{aligned}$$

Note that the interpretation of tuples implies an open world semantics for tuple types similar to the one adopted by Cardelli [24], and that $(p\uparrow)$ selects objects which do not have the path p.

It should be noted than an interpretation does not necessarily imply that the extension of a named type is identical to the type description associated with the type name via the schema σ. For this purpose, we have to further constrain the interpretation function.

Definition 2.1 *An interpretation function $\mathcal{I}$ is a* legal instance *of a schema σ iff the set $\mathcal{O}$ is finite, and for all $N \in \mathbf{N}$:*

$$\begin{aligned}
\mathcal{I}[N] &\subseteq \mathcal{I}[\sigma_P(N)] \qquad \textit{if } N \in \textit{dom } \sigma_P \\
\mathcal{I}[N] &= \mathcal{I}[\sigma_V(N)] \qquad \textit{if } N \in \textit{dom } \sigma_V
\end{aligned}$$

From the above definition, we see that the interpretation of a primitive type name *is included* in the interpretation of its description, while the interpretation of a virtual type *is* the interpretation of its description. In other words, the interpretation of a primitive type name has to be provided by the user, according to the given description, while the interpretation of a virtual type name is drawn from its definition and from the interpretation of primitive type names, thus corresponding to a view in database context. Since an $\mathcal{OLCP}$ schema is acyclic, the legal instance is uniquely determined, once the standard interpretation $\mathcal{I}_{\mathbf{B}}$ of base types and the object assignment δ have been fixed, by the interpretations of primitive type names.

It should be noted that in the our framework multiple inheritance is realized as a semantic property via the intersection operator. If we add, for example, the following type names to the schema of table 3:

$$\begin{aligned}
\sigma_P(\texttt{SeniorEmployee}) &= \texttt{Employee} \sqcap \triangle[\texttt{level}\colon [\texttt{seniority}\colon \texttt{Integer}]] \\
\sigma_P(\texttt{SeniorManager}) &= \texttt{SeniorEmployee} \sqcap \texttt{Manager}
\end{aligned}$$

we note that the class `SeniorManager` inherits from `SeniorEmployee` and `Manager`, i.e., the `SeniorManager` objects have values that satisfy the restrictions spelled out in `SeniorEmployee` and `Manager`. This means that for the `level` attribute, which is defined in both `SeniorEmployee` and `Manager`, the following restriction is met

$$\texttt{level}\colon [\texttt{seniority}\colon \texttt{Integer}, \texttt{qualification}\colon \texttt{String}, \texttt{parameter}\colon \texttt{Integer}]$$

i.e., the restrictions on the `level` attribute are simply conjunctively combined. In other words, it is not necessary to “resolve inheritance conflicts” on attributes that inherit different value ranges from multiple classes as in O_2 [7], but the value range of an attribute is simply the intersection over the value ranges of this attribute in all parent classes.

2.5 Schema Consistency

As usual, the notion of consistency is embodied by the possibility of filling a type with at least one value for some legal instance of the schema.

Definition 2.2 *Given a schema σ over $\mathbf{S}(\mathbf{A}, \mathbf{N})$, a type $S \in \mathbf{S}$ is* consistent *if and only if there is a legal instance $\mathcal{I}$ of σ and an object assignment δ, such that $\mathcal{I}[S] \neq \emptyset$. A schema σ over $\mathbf{S}(\mathbf{A}, \mathbf{N})$ is called* consistent *iff for all $N \in \mathbf{N}$, N is consistent.*

Note that for the consistency of a schema σ over $\mathbf{S}$ it is required that every type name N is consistent, but not necessarily that every type description $S \in \mathbf{S}$ is consistent, since an inconsistent type used in a set type does

not give rise to an inconsistency. In fact, $\{\bot\}_\forall$ is a consistent type denoting the empty set: $\mathcal{I}[\{\bot\}_\forall] = \{\emptyset\}$; note that $\{\bot\}_\exists$ is an inconsistent type. Of course, the consistency of a type is related to a specific schema, since the type description usually contains type names belonging the schema.

Finally, given two types S_1, S_2 of a schema σ, we say that S_1 *subsumes* S_2 iff $\mathcal{I}[S_1] \supseteq \mathcal{I}[S_2]$ for all legal instances $\mathcal{I}$ of σ. Consistency and subsumption can be reduced to each other, according to the following rules: S_1 is subsumed by S_2 iff $S_1 \sqcap \neg S_2$ is inconsistent, and S is consistent iff it is not subsumed by $\bot$. Therefore, an algorithm for checking the consistency of a type can also be used for subsumption computation.

3 Checking Consistency in $\mathcal{OLCP}$

In the following we will give an algorithm for consistency checking which is based on the rules known as *tableaux calculus* for first order logic. The algorithm can be used to determine if a type in a given schema can be filled with objects and values or, conversely, if it is always empty, because of data definitions and/or integrity constraints. It is a significant extension of the algorithm firstly developed for *DLs* in [35] as it has been necessary to develop many new rules to deal with complex value types and integrity constraints involving complex value types. The algorithm performs three steps:

1. transformation of a type into a *normal form*, that is a form where essentially complements of complex structures are eliminated (see Table 4);

2. generation, for the type in normal form, of a *constraint system*[2], which is a set of associations between variables and types;

3. examination of the constraint system to determine if it contains a *clash*: the presence of a clash is an indicator to the type inconsistency.

The execution of step 1 pushes complements of complex structures towards the innermost levels; moreover, hidden equalities and emptyset types are made explicit. This step substantially simplifies the structure of the rules implementing step 2. The normalization rewriting rules are given in Table 4.

Transformations 1-9 are obvious. Transformation 10 translates the tuple syntax into the equivalent intersection of one-attribute tuples; the equivalence is implied by the open world semantics adopted for tuples. Transformation 11 reflects the equivalence between the complement of an attribute filled with a specific type, and either the absence of the attribute, or the same attribute filled with the complement of the filler type. Transformation 12 states that the complement of an object description with a specific type is either a non-object description $\triangle\uparrow$ or an object description with the complement of the type. Transformations 13 and 14 take into account the existential and universal set specification; the type $\neg\{\top\}_\forall$ denotes all the values which are not set values. Transformation 15 reduces the complement of a predicate to either the undefinedness of one of the two involved paths or a predicate with the same paths and the negation of the comparison operator. It is important to note that the transformation of negated predicates holds because paths do not come across sets, i.e., are functional. Transformations 18-19 reduce the undefinedness of path to the undefinedness of its components.

It is easy to see that the above transformations preserve the interpretation of type descriptions. It follows that normalizing the type name descriptions of a given schema σ yields a new schema (the normal form of σ) which has the same legal instances of σ; this implies that the consistency property on the original schema σ and on its normal form are identical.

Following the method introduced in [35], we are going to devise a calculus for checking the consistency of type descriptions. The calculus will operate on syntactic entities, called constraints, consisting of variables, types and paths. Types are supposed to be in normal form.

We assume that there exists an alphabet of variable symbols, which will be denoted by the letters x, y and z.

[2]Not to be confused with the set of integrity constraints which are included in the schema definition. In the remainder of this section the term *constraint* will indicate an element of a constraint system.

$$\neg\top \longrightarrow \bot \tag{1}$$

$$\neg\bot \longrightarrow \top \tag{2}$$

$$\neg(S \sqcap S') \longrightarrow \neg S \sqcup \neg S' \tag{3}$$

$$\neg(S \sqcup S') \longrightarrow \neg S \sqcap \neg S' \tag{4}$$

$$\neg\neg S \longrightarrow S \tag{5}$$

$$p \leq d \longrightarrow p < d \sqcup p = d \tag{6}$$

$$p \geq d \longrightarrow p > d \sqcup p = d \tag{7}$$

$$p_1 \leq p_2 \longrightarrow p_1 < p_2 \sqcup p_1 = p_2 \tag{8}$$

$$p_1 \geq p_2 \longrightarrow p_1 > p_2 \sqcup p_1 = p_2 \tag{9}$$

$$[a_1 \colon S_1, \cdots, a_p \colon S_p] \longrightarrow [a_1 \colon S_1] \sqcap \cdots \sqcap [a_p \colon S_p] \tag{10}$$

$$\neg[a \colon S] \longrightarrow a{\uparrow} \sqcup [a \colon \neg S] \tag{11}$$

$$\neg \triangle S \longrightarrow (\triangle{\uparrow}) \sqcup \triangle\neg S \tag{12}$$

$$\neg\{S\}_\exists \longrightarrow \neg\{\top\}_\forall \sqcup \{\neg S\}_\forall \tag{13}$$

$$\neg\{S\}_\forall \longrightarrow \neg\{\top\}_\forall \sqcup \{\neg S\}_\exists \text{ if } S \neq \top \tag{14}$$

$$\neg(p_1 \ \theta \ p_2) \longrightarrow (p_1{\uparrow}) \sqcup (p_2{\uparrow}) \sqcup (p_1 \overline{\theta} p_2) \tag{15}$$

$$(e \,.\, p\theta d) \longrightarrow \begin{cases} [a : (p\theta d)], & \text{if } e = a \in \mathbf{A} \\ \triangle(p\theta d), & \text{if } e = \triangle \end{cases} \tag{16}$$

$$(e\theta d) \longrightarrow \begin{cases} [a : (\epsilon\theta d)], & \text{if } e = a \in \mathbf{A} \\ \triangle(\epsilon\theta d), & \text{if } e = \triangle \end{cases} \tag{17}$$

$$(e \,.\, p{\uparrow}) \longrightarrow (e{\uparrow}) \sqcup \begin{cases} [a : (p{\uparrow})], & \text{if } e = a \in \mathbf{A} \\ \triangle(p{\uparrow}), & \text{if } e = \triangle \end{cases} \tag{18}$$

$$\neg(e{\uparrow}) \longrightarrow \begin{cases} [a : \top], & \text{if } e = a \in \mathbf{A} \\ \triangle\top, & \text{if } e = \triangle \end{cases} \tag{19}$$

$$\neg(\epsilon\theta d) \longrightarrow \epsilon\overline{\theta} d \sqcup \neg B \qquad \text{where } d \in B \tag{20}$$

$$(\epsilon{\uparrow}) \longrightarrow \bot \tag{21}$$

Table 4: Normalization rewriting rules

Definition 3.1 *A* constraint *is a syntactic expression of the form:*

- x/S, *where* S *is a type in normal form*
- xpy
- $x\exists y$
- $x\forall y$
- $x\theta y$, *where* $\theta \in \{=, \neq, <, >\}$
- $x = \{d_1, \ldots, d_n\}$, *with* $d_i \in \mathcal{D}$, *for* $i \in 1..n$.

Let $\mathcal{I}$ be a legal instance. An *$\mathcal{I}$–assignment* is a function α that maps every variable to an element of $\mathcal{V}$. We say that:

- α *satisfies* x/S, if $\alpha(x) \in \mathcal{I}[S]$
- α *satisfies* xpy, if $(\alpha(x), \alpha(y)) \in \mathcal{J}[p]$
- α *satisfies* $x\exists y$, if $\alpha(x) = \{\ldots, v, \ldots\}$, and $\alpha(y) = v$
- α *satisfies* $x\forall y$, if $\alpha(x) = \{\ldots, v, \ldots\}$, and $\alpha(y) = v$
- α *satisfies* $x\theta y$, if $\alpha(x)\theta\alpha(y)$ is true
- α *satisfies* $x = \{d_1, \ldots, d_n\}$, if $\alpha(x) = \{d_1, \ldots, d_n\}$.

A constraint κ is *satisfiable* if there is a legal instance $\mathcal{I}$ and an *$\mathcal{I}$–assignment* α such that α satisfies κ. A *constraint system* $\mathbf{K}$ is a finite set of constraints. An *$\mathcal{I}$–assignment* α *satisfies* a constraint system $\mathbf{K}$ if α satisfies every constraint in $\mathbf{K}$. A constraint system $\mathbf{K}$ is *satisfiable* if there is a legal instance $\mathcal{I}$ and an *$\mathcal{I}$–assignment* α such that α satisfies $\mathbf{K}$.

The satisfiability of a constraint system is related to the consistency of a type: a normal type S is consistent if and only if the constraint system $\{x/S\}$ is satisfiable.

First of all, let us consider base types and comparisons involving base types.

Definition 3.2 *Let* $\mathbf{K}_b$ *be a constraint system. We say that* $\mathbf{K}_b$ *is a* base *constraint system if and only if* $\kappa \in \mathbf{K}_b$ *implies* κ *is of one of the following forms:*

1. x/B
2. $x/(\epsilon\theta d)$, $\theta \in \{=, <, >\}$
3. $x\theta y$, $\theta \in \{<, >\}$
4. $x/(\epsilon \neq d)$, *where* x *occurs in at least one constraint of the form 1, 2, 3.*
5. $x/\neg B$, *where* x *occurs in at least one constraint of the forms 1, 2, 3.*
6. $x \neq y$, *where* x *and* y *occur in at least one constraint of the forms 1, 2, 3.*

The satisfiability of a base constraint system has already been dealt with in the literature (see, for example, [45]). We denote by $R(x, \mathbf{K})$ the set of base values which satisfy all the constraints of the form 1, 2, 4, 5 in $\mathbf{K}$. Notice that the finiteness of $R(x, \mathbf{K})$, and $R(x, \mathbf{K})$ itself, can be computed in polynomial time. Let us consider now the satisfiability for a type S. The calculus starts with a constraint system $\mathbf{K} = \{x/S\}$ and adds constraints to $\mathbf{K}$ until either a contradiction is generated or an interpretation satisfying S can be obtained from the resulting system. With $\mathbf{K}_z^y$ we denote the constraint system obtained from $\mathbf{K}$ by replacing each occurrence of z by y. The calculus is given by the *completion rules* shown in Table 5.

Rules 1, 6, 9, 10 and 11 work on set constructors. Rule 2: Part 2a is obvious; part 2b reflects the semantics of ϵ; part 2c reflects the fact that attributes are interpreted as functions and δ is a function. Rule 3 works on

1. $\mathbf{K} \longrightarrow_{\emptyset} \{x = \emptyset\} \cup \mathbf{K}$ if $x/\{\bot\}_{\forall}$ is in $\mathbf{K}$, and $x = \emptyset$ is not in $\mathbf{K}$.

2. $\mathbf{K} \longrightarrow_{replace} \mathbf{K}^{y}_{z}$ if
   (a) $y = z$, or
   (b) $y \epsilon z$, or
   (c) xey and xez, with $e \in \mathbf{A} \cup \{\triangle, \forall\}$
   are in $\mathbf{K}$, and y and z are not the same variable.

3. $\mathbf{K} \longrightarrow_{\sigma} \{x/\sigma(N)\} \cup \mathbf{K}$ if $x/N \in \mathbf{K}$ and $x/\sigma(N) \notin \mathbf{K}$.

4. $\mathbf{K} \longrightarrow_{\neg} \{x/\neg\sigma_V(N)\} \cup \mathbf{K}$ if $x/\neg N \in \mathbf{K}$ and $x/\neg\sigma_V(N) \notin \mathbf{K}$.

5. $\mathbf{K} \longrightarrow_{\neq} \{x \neq y\} \cup \mathbf{K}$ if $x/N \in \mathbf{K}$ and $y/\neg N \in \mathbf{K}$ and none of $x \neq y$, $y \neq x$ is in $\mathbf{K}$.

6. $\mathbf{K} \longrightarrow_{\exists\forall} \{y/S\} \cup \mathbf{K}$ if $x\exists y$ and $x/\{S\}_{\forall}$ is in $\mathbf{K}$, and y/S is not in $\mathbf{K}$.

7. $\mathbf{K} \longrightarrow_{\sqcap} \{x/S_1, x/S_2\} \cup \mathbf{K}$ if $x/S_1 \sqcap S_2 \in \mathbf{K}$, and x/S_1 and x/S_2 are not both in $\mathbf{K}$.

8. $\mathbf{K} \longrightarrow_{\sqcup} \{x/S_3\} \cup \mathbf{K}$ if $x/S_1 \sqcup S_2 \in \mathbf{K}$, neither x/S_1 nor x/S_2 is in $\mathbf{K}$, and $S_3 = S_1$ or $S_3 = S_2$.

9. $\mathbf{K} \longrightarrow_{\{d\}} \{x = \{d_0, \ldots, d_n\}\} \cup \mathbf{K}$ if $x/\{S\}_{\forall}$, $x\forall y$ are in $\mathbf{K}$, $R(y, \mathbf{K})$ is finite, $d_i \in R(y, \mathbf{K})$ for all $i \in 0..n$, and there is no constraint of the form $x = \{d'_0, \ldots, d'_m\}$ in $\mathbf{K}$.

10. $\mathbf{K} \longrightarrow_{\forall} \mathbf{K}' \cup \mathbf{K}$ if $x/\{S\}_{\forall}$ is in $\mathbf{K}$, and $\mathbf{K}'$ is either $\{x = \emptyset\}$ or $\{x\forall y, y/S\}$, and $x = \emptyset$ is not in $\mathbf{K}$ and there is no variable z such that $x\forall z$, z/S are in $\mathbf{K}$, and y is a new variable.

11. $\mathbf{K} \longrightarrow_{\exists} \{x\exists y, y/S\} \cup \mathbf{K}$ if $x/\{S\}_{\exists} \in \mathbf{K}$, and there is no variable z such that $x\exists z, z/S$ are in $\mathbf{K}$, and y is a new variable.

12. $\mathbf{K} \longrightarrow_{[]} \{xay, y/S\} \cup \mathbf{K}$ if $x/[a\colon S] \in \mathbf{K}$, and there is no variable z such that $xaz, z/S$ are in $\mathbf{K}$, and y is a new variable.

13. $\mathbf{K} \longrightarrow_{\triangle} \{x \triangle y, y/S\} \cup \mathbf{K}$ if $x/\triangle S \in \mathbf{K}$, and there is no variable z such that $x \triangle y$, y/S is in $\mathbf{K}$, and y is a new variable.

14. $\mathbf{K} \longrightarrow_{\theta} \{xp_1y, xp_2z, (y\theta z)\} \cup \mathbf{K}$ if $x/(p_1\theta p_2) \in \mathbf{K}$, and there are no variables y', z' such that xp_1y', xp_2z' and $(y'\theta z')$ are in $\mathbf{K}$, and y and z are new variables.

15. $\mathbf{K} \longrightarrow_{path} \{xez, zpy\} \cup \mathbf{K}$ if $xe.py \in \mathbf{K}$, there is no variable z' such that xez' and $z'py$ are in $\mathbf{K}$, and z is a new variable.

Table 5: Completion rules

type names, by reflecting the fact that a type name N is consistent only if its description is consistent, thus it adds the constraint $y/\sigma(N)$ to the constraint system. Rule 4: the property that the complement of type names N, $\neg N$ is consistent only if the complement of its description is consistent holds only for $N \in \text{dom } \sigma_V$. Rule 5 reflects the fact that N is disjoint from $\neg N$, thus it adds the constraint $x \neq y$ to the constraint system. Rules 7 and 8 imitate the tableaux rules for intersections and unions, respectively. Rule 12 works on tuple constructors, by reflecting the fact that the type $[a : S]$ is consistent only if the type S is consistent, thus it adds the constraint y/S to the constraint system. In addition, the constraint xay can fire rule 2, in order to replace variable names deriving from other tuple constraints or from atomic predicates with a path including the attribute a. Rule 13 works on $\triangle$ constructor: it reflects the fact that the type $\triangle S$ is consistent only if the type S is consistent, therefore it adds y/S to the constraint system. In addition, the constraint $x \triangle y$ can fire rule 2, in order to replace variable names deriving from other object constraints or from atomic predicates with a path including the symbol $\triangle$. Rule 14 works on path equations. It produces path constraints of the form xpy, that are stepwise shortened by rule 15 by stripping off the first symbol. It should be observed that rules 8, 9 and 10 are *nondeterministic*, since one of the alternatives must be chosen. Moreover, rules from 10 on introduce new variables in the constraint system.

Definition 3.3 *Let κ_x denote a base constraint in which the variable x occurs. A* clash *is a set of constraints of one of the following forms:*

1. *a base constraint system that is not satisfiable;*
2. $\{\kappa_x, xey\}$, *with* $e \in \mathbf{A} \cup \{\triangle, \exists, \forall\}$;
3. $\{x \neq x\}$;
4. $\{x/\bot\}$;
5. $\{x/N, x/\neg N\}$;
6. $\{x/\sigma(N), x/\neg N\}$, *with* $N \in$ *dom* σ_V;
7. $\{xey, x/(e\uparrow)\}$, *with* $e \in \mathbf{A} \cup \{\triangle\}$;
8. $\{xey, xe'y'\}$, *where* $(e, e') \notin \mathbf{A}^2 \cup \{\triangle\}^2 \cup \{\exists, \forall\}^2$;
9. $\{x/\{S\}_*, x/\neg\{\top\}_\forall\}$, *where* $*$ *denotes both* $\forall$ *and* $\exists$;
10. $\{x = \emptyset, xey\}$, *with* $e \in \mathbf{A} \cup \{\triangle, \exists, \forall\}$;
11. $\{x = \emptyset, \kappa_x\}$;
12. $\{x_i e_i x_{i+1} \mid 1 \leq i \leq n-1, e_i \neq \triangle\}$, *with* $x_1 = x_n$;
13. $\{x \neq y, x = v, y = v\}$, *with* $v = \{d_1, \ldots, d_n\}$, $n \geq 0$;
14. $\{x = \{d_0, \ldots, d_n\}, x\forall y\}$ *with* $d_i \notin R(y, \mathbf{K})$, *for some* $i \in 0..n$;
15. $\{x = \{d_0, \ldots, d_n\}, x\exists y\}$ *with* $d_i \notin R(y, \mathbf{K})$, *for all* $i \in 0..n$.

We now give a few examples of the less obvious clashes. As an example of clash 2, let κ_x be the base constraint $x/\texttt{String}$; then $\{\kappa_x, x \triangle y\}$ is unsatisfiable because we should assign to x a value which is at same time a string and an object identifier. Clash 8 reflects the fact that the set of constraints $\{xay, x \triangle z\}$ (which derives from a type of the form $[a\colon S_1] \sqcap \triangle S_2$) is unsatisfiable because we should assign to x a value which is at same time a tuple and an object identifier. As an example of 12, let us consider $\{xax\}$, which derives, for example, from the type $[a\colon S] \sqcap (\epsilon = a)$. An *$\mathcal{I}$–assignment* α that satisfies xax, i.e., such that $(\alpha(x), \alpha(x)) \in \mathcal{J}[a]$, should require that $\alpha(x) = [a\colon \alpha(x), \ldots]$, but this cannot hold because all values in $\mathcal{V}$ are only finitely nested. On the other hand, $\{x \triangle x\}$, is not a clash, because an *$\mathcal{I}$–assignment* α with $\delta(\alpha(x)) = \alpha(x)$, satisfies it.

A constraint system containing a clash is obviously unsatisfiable. A constraint system is *complete* if none of the rules applies to it, and *incomplete* otherwise.

Let $\mathbf{K}_1$, $\mathbf{K}_2$, ..., $\mathbf{K}_n$ be a sequence of constraint systems such that $\mathbf{K}_i \rightarrow_* \mathbf{K}_{i+1}$ is an instance of a completion rule and $\mathbf{K}_n$ is complete. Then we say that $\mathbf{K}_n$ is a *completion* of $\mathbf{K}_1$, and $\mathbf{K}_i$ is a *partial completion* of $\mathbf{K}_1$.

The main properties of the calculus are stated by the following theorems, where each type is assumed to be in normal form. The proofs for the theorems are given in Appendix A.1.

Theorem 3.4 (Termination) *Let S be a $\mathcal{OLCP}$ type. There is no infinite sequence of constraint systems $\mathbf{K}_1 = \{x/S\}$, $\mathbf{K}_2$, ..., such that $\mathbf{K}_i \rightarrow_* \mathbf{K}_{i+1}$, for $i \geq 1$, is an instance of a completion rule.*

Theorem 3.5 (Completeness) *Let S be a $\mathcal{OLCP}$ type. Then S is consistent iff some completion of $\{x/S\}$ contains no clash.*

Since every nondeterministic rule can generate at most a finite number of guesses, the above theorems entail that the calculus can be turned into an algorithm: To check a type S for consistency, we generate all the complete constraint systems which can be derived from $\{x/S\}$. If all the generated systems contain a clash, then S is inconsistent, otherwise it is consistent.

Example As an example of application of the calculus for consistency checking, let us now consider the satisfiability for the type associated to the class `BossTechnician`:

$$\begin{aligned}\overline{S} \quad = \quad & (\epsilon = \triangle.\texttt{works}.\ \triangle\ .\texttt{manager})\sqcap \\ & (\triangle.\texttt{salary} < \triangle.\texttt{works}.\ \triangle\ .\texttt{manager}.\ \triangle\ .\texttt{salary})\end{aligned}$$

Let $\mathbf{K} = \{x/\overline{S}\}$ be the initial constraint system; a sequence of applications of the completion rules to $\mathbf{K}$ is as follows:

$\mathbf{K} := \mathbf{K} \cup \{x/(\epsilon = \triangle.\texttt{works}.\ \triangle\ .\texttt{manager}),$

$\quad x/(\triangle.\texttt{salary} < \triangle.\texttt{works}.\ \triangle\ .\texttt{manager}.\ \triangle\ .\texttt{salary})\}$ $\qquad (\rightarrow_\sqcap)$

$\mathbf{K} := \mathbf{K} \cup \{x \epsilon x_1, x\ \triangle\ .\texttt{works}.\ \triangle\ .\texttt{manager}\ x_2, x_1 = x_2\}$ $\qquad (\rightarrow_\theta)$

$\mathbf{K} := \mathbf{K} \cup \{x\ \triangle\ .\texttt{salary}\ x_3, x\ \triangle\ .\texttt{works}.\ \triangle\ .\texttt{manager}.\ \triangle\ .\texttt{salary}\ x_4, x_3 < x_4\}$ $\qquad (\rightarrow_\theta)$

$\mathbf{K} := \mathbf{K} \cup \{x\ \triangle\ x_5, x_5 \texttt{works}\ x_6, x_6\ \triangle\ x_7, x_7\ \texttt{manager}\ x_2, x\ \triangle\ x_8, x_8\ \texttt{salary}\ x_3,$

$\quad x\ \triangle\ x_9, x_9\ \texttt{works}\ x_{10}, x_{10}\ \triangle\ x_{11}, x_{11}\ \texttt{manager}\ x_{12}, x_{12}\ \triangle\ x_{13}, x_{13}\ \texttt{salary}\ x_4\}$ $\qquad$ (repeated application of $\rightarrow_{path}$)

$\mathbf{K} := \mathbf{K}^{x}_{x_1} = \{x \epsilon x, x = x_2, \dots\}$ $\qquad (\rightarrow_{replace}$ due to $x \epsilon x_1)$

$\mathbf{K} := \mathbf{K}^{x}_{x_2} = \{x = x, x_7\ \texttt{manager}\ x, \dots\}$ $\qquad (\rightarrow_{replace}$ due to $x = x_2)$

$\mathbf{K} := \mathbf{K}^{x_5}_{x_8} = \{x\ \triangle\ x_5, x_5\ \texttt{salary}\ x_3, \dots\}$ $\qquad (\rightarrow_{replace}$ due to $x\ \triangle\ x_8, x\ \triangle\ x_5)$

$\mathbf{K} := \mathbf{K}^{x_5}_{x_9} = \{x\ \triangle\ x_5, x_5 \texttt{works} x_{10}, \dots\}$ $\qquad (\rightarrow_{replace}$ due to $x\ \triangle\ x_9, x\ \triangle\ x_5)$

$\mathbf{K} := \mathbf{K}^{x_6}_{x_{10}} = \{x_5\ \texttt{works}\ x_6, x_6\ \triangle\ x_{11}, \dots\}$ $\qquad (\rightarrow_{replace}$ due to $x_5\ \texttt{works}\ x_6, x_5\ \texttt{works}\ x_{10})$

$\mathbf{K} := \mathbf{K}^{x_7}_{x_{11}} = \{x_6\ \triangle\ x_7, x_7\ \texttt{manager}\ x_{12}, \dots\}$ $\qquad (\rightarrow_{replace}$ due to $x_6\ \triangle\ x_{11}, x_6\ \triangle\ x_7)$

$\mathbf{K} := \mathbf{K}^{x}_{x_{12}} = \{x_7\ \texttt{manager}\ x, x\ \triangle\ x_{13}, \dots\}$ $\qquad (\rightarrow_{replace}$ due to $x_7\ \texttt{manager}\ x_{12}, x_7\ \texttt{manager}\ x)$

$\mathbf{K} := \mathbf{K}^{x_5}_{x_{13}} = \{x\ \triangle\ x_5, x_5\ \texttt{salary}\ x_4, \dots\}$ $\qquad (\rightarrow_{replace}$ due to $x\ \triangle\ x_{13}, x\ \triangle\ x_5)$

$\mathbf{K} := \mathbf{K}^{x_3}_{x_4} = \{x_5\ \texttt{salary}\ x_3, x_3 < x_3, \dots\}$ $\qquad (\rightarrow_{replace}$ due to $x_5\ \texttt{salary}\ x_4, x_5\ \texttt{salary}\ x_3)$

One can verify that no rule can be applied to $\mathbf{K}^{x_3}_{x_4}$; since it contains the constraint $x_3 < x_3$, which is obviously unsatisfiable (namely it is a clash), the type $\overline{S}$ is inconsistent.

3.1 Complexity bounds

It is can be easily seen that the algorithm devised above may require exponential space. In fact, if we let:

$$S_0 = S'_0$$
$$S_{n+1} = [a\colon \{S'_{n+1}\}_\exists \sqcap \{S''_{n+1}\}_\exists \sqcap \{S_n\}_\forall]$$

where S'_n, S''_n are arbitrary types for any $n \geq 0$, then some constraint system issued from x/S_n contains a number of variables which is exponential in n. Quite clearly, this explosion is due to the interaction between nested existential and universal set specifications. This interplay between quantifiers occurs also in any description logic containing both existentially and universally quantified roles (see e.g. [30]). Since $\mathcal{OLCP}$ contains general complements, it may be argued that the consistency problem $\mathcal{OLCP}$ is at least as difficult as the satisfiability problem in $\mathcal{ALC}$, which is PSPACE-complete. Indeed the following result holds:

Theorem 3.6 *The consistency problem for $\mathcal{OLCP}$ types is PSPACE-hard.*

However, the interplay of the quantifiers is not the only source of complexity. The $\rightarrow_{\{d\}}$-rule introduces constraints of the form $x = \{d_1, \ldots, d_n\}$, which are not bounded by any expression in the schema, but, rather, by the numbers representing base values occurring in range restrictions. In fact, the number of base value designators between braces in the above constraint can be as high as $|R(x, \mathbf{K})|$. Therefore, the above given decidability result implies that the consistency problem for $\mathcal{OLCP}$ types can be decided with space bounded by the product of an exponential in the input length and the greatest integer occurring in the problem instance.

We conjecture that a tighter upper bound could be achieved by using the method of *traces*, originally applied to the description logic $\mathcal{ALC}$ in [46]. The method essentially exploits the syntactic properties of constraint systems obtained using completion rules to keep in memory only a portion of a constraint system at a time. A modified set of rules, the *trace rules*, are obtained from the calculus by replacing the $\rightarrow_\exists$-rule with a more constrained version ($\rightarrow_{T\exists}$), defined as follows:

11. $\mathbf{K} \longrightarrow_{T\exists} \{x\exists y, y/S\} \cup \mathbf{K}$ if $x/\{S\}_\exists \in \mathbf{K}$, and there is no variable z such that $x\exists z$ is in $\mathbf{K}$, and y is a new variable.

A *trace* is a constraint system built by *trace rules*, by applying rules 6 and 11 only if no other rule is applicable. Our conjecture is that a complete constraint system $\mathbf{K}$ can then be obtained as the union of finitely many traces, and that $\mathbf{K}$ contains a clash if and only if some of its traces contains a clash. Since the cardinality of a trace is linearly bounded, the consistency problem for $\mathcal{OLCP}$ types could thus be decided with space polynomial in the product of the input length and the greatest integer occurring in the problem instance.

4 Evaluating extensions of $\mathcal{OLCP}$

As has been discussed in section 1, there are many directions of possible extensions to the kernel model $\mathcal{OLC}$, which have been considered, but our main guideline was decidability. For this reason we start from the model $\mathcal{OLCP}$ presented in section 2 and consider two possible independent and relevant extensions: cyclic type names and paths across multi-valued attributes. Unfortunately, as it will be proved in this section, both these extensions lead to undecidability.

4.1 Extending schemata with cyclic type names

Let us suppose that cyclic type names, i.e., type names which make direct or indirect references to themselves through attributes are allowed. The capability of giving cyclic descriptions adds additional expressive power to CODMs; a classical example of a cyclic class together with integrity constraint is the following: an employee is managed by an employee and earns not more than its manager:

$$\underline{\textbf{class}}\ \texttt{Employee} \quad = \quad [\texttt{name}\colon \texttt{String}, \texttt{salary}\colon \texttt{Integer}, \texttt{boss}\colon \texttt{Employee}]$$
$$\underline{\textbf{and}}\ (\texttt{salary} \leq \texttt{boss.salary})$$

By the description above, an employee cannot earn more than *all* its managers, i.e., managers at any level, and, in particular, the equal sign holds for bosses, which are managers of themselves.

The inclusion of virtual cyclic type names in a schema gives rise to the new problem of choosing one of the possible semantics to interpret cyclic definitions: *least fixpoint*, *greatest fixpoint* or *descriptive* semantics [10, 43]. As a consequence, the interpretation of a virtual cyclic type name is not uniquely determined by the interpretations of primitive type names. For example, the class of employees who earn less than \$10,000 and are managed by employees with the same limitation, is described by the following cyclic virtual class:

$$\underline{\textbf{virtual class}}\ \texttt{EmployeeV} \quad = \quad \underline{\textbf{isa}}\ \texttt{Employee}\ \underline{\textbf{and}}\ [\texttt{boss}\colon \texttt{EmployeeV}]\ \ \underline{\textbf{and}}\ (\texttt{salary} \leq 10)$$

For a given interpretation of the primitive class `Employee` there may be several ways to interpret `EmployeeV` in such a way that the above equality holds, depending on whether we choose *least fixpoint*, *greatest fixpoint* or *descriptive* semantics. It is beyond the scope of this paper to discuss the impact of the various semantics. Let us consider the less restrictive, i.e., the descriptive semantics, which accepts any interpretation which is *a model* of the schema. With these hypotheses we prove in Appendix B the following result:

Theorem 4.1 (Cycles and undecidability) *Checking consistency in a cyclic $\mathcal{OLCP}$ schema interpreted under descriptive semantics is an undecidable problem.*

It is worth note that the descriptive semantics coincides with the one usually adopted in a cyclic schema with only primitive type names. Thus the above theorem holds also for schemata without virtual type names. The above result shows that if we want a decidable consistency checking algorithm we cannot allow cyclic $\mathcal{OLCP}$ schemata.

The proof of theorem 4.1 essentially uses a cyclic type name together with path relations. On the other hand, [29] shows that the satisfiability problem for a description logic comprising general complements and general inclusion statements (which subsume terminological cycles) is solvable. Therefore the question arises whether the consistency problem for cyclic schemata without path relations interpreted under descriptive semantics is solvable.

In the following section, the language $\mathcal{OLCD}$ allowing cyclic definitions (but not path relations) and calculus solving its consistency problem, which derives from the calculus for $\mathcal{OLCP}$, are presented.

4.2 $\mathcal{OLCD}$: cyclic schemata without path relations

In the sequel, we briefly list the differences in the $\mathcal{OLCD}$ syntax, semantics and calculus, with respect to $\mathcal{OLCP}$.

Types and Schemas In $\mathcal{OLCD}$, $p\theta p'$ is eliminated from the syntax of the type system and the condition of acyclic schema is relaxed; on the other hand,the restriction on nesting of set types holds. Notice that $\mathcal{OLCD}$ schemata need not be inheritance well-founded, and we do not impose such a property.

Interpretation and Database Instances We consider the descriptive semantics which accepts as legal istance of a schema σ any interpretation satisfying the relations of definition 2.1 ($\mathcal{I}[N] \subseteq \mathcal{I}[\sigma_P(N)]$ if $N \in$ dom σ_P, $\mathcal{I}[N] = \mathcal{I}[\sigma_V(N)]$ if $N \in$ dom σ_V). For instance, let us consider the classes `Employee` (without the path relation ($\texttt{salary} \leq \texttt{boss.salary}$)) and `EmployeeV` of the previous subsection:

$$\begin{aligned} \sigma_P(\texttt{Employee}) &= [\texttt{name}\colon \texttt{String}, \texttt{salary}\colon \texttt{Integer}, \texttt{boss}\colon \texttt{Employee}] \\ \sigma_V(\texttt{EmployeeV}) &= \texttt{Employee} \sqcap \triangle[\texttt{boss}\colon \texttt{EmployeeV}] \sqcap (\triangle.\texttt{salary} \leq 10) \end{aligned}$$

and the following assignments of values to objects:

$$\delta\colon \begin{cases} o_1 \mapsto [\texttt{name}\colon \textit{"Robert"}, \texttt{salary}\colon 8, \texttt{boss}\colon o_2] \\ o_2 \mapsto [\texttt{name}\colon \textit{"Jean"}, \texttt{salary}\colon 11, \texttt{boss}\colon o_3] \\ o_3 \mapsto [\texttt{name}\colon \textit{"Joel"}, \texttt{salary}\colon 9, \texttt{boss}\colon o_3] \\ o_4 \mapsto [\texttt{name}\colon \textit{"John"}, \texttt{salary}\colon 8, \texttt{boss}\colon o_3] \\ o_5 \mapsto [\texttt{name}\colon \textit{"Lothar"}, \texttt{salary}\colon 7, \texttt{boss}\colon o_5] \\ \vdots \end{cases}$$

If we fix $\mathcal{I}[\texttt{Employee}] = \{o_1, o_2, o_3, o_4, o_5\}$, then a legal istance of σ is obtained by choosing for $\mathcal{I}[\texttt{EmployeeV}]$ one of the following sets: $\{o_3, o_4, o_5\}$, $\{o_3, o_4\}$, $\{o_5\}$, $\emptyset$.

Schema Consistency Cyclic type names introduce new sources of inconsistency. Notice that, since all values in $\mathcal{V}$ are only finitely nested, a cycle in the definition of a type name must use at least one of the constructors $\triangle$, $\{\}_\forall$. For instance, if $\sigma(N) = [a\colon N]$ then N is inconsistent; however, if $\sigma(N) = \triangle[a\colon N]$ then, for any finite $\mathcal{O}$ and injective map $f\colon \mathcal{O} \to \mathcal{O}$, an interpretation such that: $\mathcal{I}[N] = \mathcal{O}$, $\delta(o) = [a : f(o)]$ for all $o \in \mathcal{O}$, is a legal instance of σ, and if $\sigma(N) = [a\colon \{N\}_\forall]$ then an interpretation such that $\mathcal{I}[N] = \bigcup_{i<n}\{v_i\}$, where $v_0 = [a : \emptyset]$, $v_i = [a : \{v_{i-1}\}]$, for any finite $n > 1$, is a legal instance of σ.

Checking Consistency First, we can exclude the normalization rewriting rules dealing with path relations, i.e. rules 8, 9 and 15 of table 4. Moreover, in $\mathcal{OLCD}$ only a subset of the constraints defined in 3.1 are needed. More precisely: in xpy, p is a path of length 1, i.e., $p = e$, where $e \in \{\triangle\} \cup \mathbf{A}$ and in $x\theta y$, θ is $\neq$. As a consequence, a base constraint system does not include any constraint of the form $x\theta y$, where $\theta \in \{<, >\}$ (see definition 3.1, item 3).

We now modify completion rules. In table 5, we can drop completion rules 14 and 15 dealing with path relations. Moreover, in rule 2, part 2a and part 2b are useless because a string of the form $x = z$ or $x\epsilon z$ is not a constraint (see above), while definition 3.3 holds unchanged for $\mathcal{OLCD}$.

A major change applies to the control strategy. The current strategy only checks whether given constraints are in the system, but it can be easily seen that this does not suffice to ensure termination, if cyclic type names are allowed. For instance, if $\sigma(N) = \triangle N$, then for any finite $n \geq 0$, the constraint system $\{x_i/N, x_i/\triangle N, x_i \triangle x_{i+1} : 0 \leq i \leq n\}$ derives from $\{x_0/N\}$. We adopt a method similar to the one proposed in [29], and define the successor and predecessor relations among variables as follows.

Definition 4.2 *Let x, y be variables, $\mathbf{K}$ a constraint system. Then x is a* direct successor *of y in $\mathbf{K}$ iff $yex \in \mathbf{K}$, for some $e \in \mathbf{A} \cup \{\triangle, \exists, \forall\}$. We denote by* successor *the transitive closure of the relation direct successor and we denote by* predecessor *its inverse.*

Given a constraint system $\mathbf{K}$ and a variable x, we define a function $\Theta(\cdot,\cdot)$ by $\Theta(\mathbf{K}, x) = \{S \mid x/S \in \mathbf{K}\}$. We say that two variables x and y are *K–equivalent*, written $x \equiv_K y$, if $\Theta(\mathbf{K}, x) = \Theta(\mathbf{K}, y)$. Intuitively, two K–equivalent variables x and y can represent the same element in the potential interpretation built by the completion rules, unless $x \neq y \in \mathbf{K}$.

Based on K–equivalence, we add a new condition to the $\rightarrow_{replace}$-rule:

2. $\mathbf{K} \longrightarrow_{replace} \mathbf{K}^y_z$ if
   (a) xey and xez, with $e \in \mathbf{A} \cup \{\triangle, \forall\}$ are in $\mathbf{K}$, or
   (b) y is a predecessor of z in $\mathbf{K}$, $y \equiv_K z$

   and y and z are not the same variable.

Given a constraint system, more than one rule might be applicable to it. We define the following *strategy* for the application of rules:

1. apply a rule different from the $\rightarrow_{replace}$-rule only if this rule is not applicable;
2. apply a rule to a variable x only if no rule is applicable to a predecessor of x.

As an immediate example, let us recall the schema $\sigma(N) = [a\colon N]$. Then the initial constraint system $\{x/N\}$ yields

$$\{x/N, x/[a\colon N], xay, y/N, y/[a\colon N]\}.$$

If the strategy is not applied, the above system is incomplete and the rules would not necessarily bring it to a finite complete extension ($\rightarrow_{[]}$ and $\rightarrow_\sigma$ are repeatedly applied). Applying the strategy, we are forced to apply

the $\rightarrow_{replace}$-rule (which is applicable since $x \equiv_K y$) before any other rule, immediately obtaining the complete system

$$\{x/N, x/[a\colon N], xax\}.$$

In the sequel, we assume that rules are always applied according to this strategy and that we always start with a constraint system $\mathbf{K}$ issued from $\{x/S\}$. Under these assumptions, the properties of the calculus for $\mathcal{OLCP}$ hold for $\mathcal{OLCD}$ as well.

Theorem 4.3 (Termination) *Let S be a $\mathcal{OLCD}$ type. There is no infinite sequence of constraint systems $\mathbf{K}_1 = \{x/S\}$, $\mathbf{K}_2$, ... , such that $\mathbf{K}_i \rightarrow_* \mathbf{K}_{i+1}$, for $i \geq 1$, is an instance of a completion rule.*

Theorem 4.4 (Completeness) *Let S be a $\mathcal{OLCD}$ type. Then S is consistent iff some completion of $\{x/S\}$ contains no clash.*

Example An example of application follows. Notice the impact of the strategy on the applicability of rules.

$$\sigma_P(\texttt{Clerk}) = \triangle[\texttt{boss}\colon \texttt{Clerk}] \sqcap \triangle[\texttt{salary}\colon (\epsilon < 4)] \sqcap \triangle[\texttt{boss}\colon \triangle[\texttt{salary}\colon (\epsilon > 6)]]$$

Let $\mathbf{K} = \{x/\texttt{Clerk}\}$ be the initial constraint system; a sequence of applications of the completion rules to $\mathbf{K}$ is as follows:

$\mathbf{K} := \mathbf{K} \cup \{x/ \triangle[\texttt{boss}\colon \texttt{Clerk}] \sqcap \triangle[\texttt{salary}\colon (\epsilon < 4)] \sqcap \triangle[\texttt{boss}\colon \triangle[\texttt{salary}\colon (\epsilon > 6)]]\}$ $(\rightarrow_\sigma)$

$\mathbf{K} := \mathbf{K} \cup \{x/ \triangle[\texttt{boss}\colon \texttt{Clerk}], x/ \triangle[\texttt{salary}\colon (\epsilon < 4)], x/ \triangle[\texttt{boss}\colon \triangle[\texttt{salary}\colon (\epsilon > 6)]]\}$ $(\rightarrow_\sqcap)$

No more rules are applicable to x. We may now apply rules to the successors x_1 of x:

$\mathbf{K} := \mathbf{K} \cup \{x \triangle x_1, x_1/[\texttt{boss}\colon \texttt{Clerk}], x_1/[\texttt{salary}\colon (\epsilon < 4)], x_1/[\texttt{boss}\colon \triangle[\texttt{salary}\colon (\epsilon > 6)]]\}$ $(\rightarrow_\triangle$ and $\rightarrow_{replace})$

No more rules are applicable to x_1; we may apply rules to successors of x_1:

$\mathbf{K} := \mathbf{K} \cup \{x_1 \texttt{boss} x_2, x_2/\texttt{Clerk}, x_1 \texttt{salary} x_3, x_3/(\epsilon < 4), x_2/ \triangle[\texttt{salary}\colon (\epsilon > 6)]\}$ $(\rightarrow_{[]}$–and $\rightarrow_{replace})$

$\mathbf{K} := \mathbf{K} \cup \{x_2/ \triangle[\texttt{boss}\colon \texttt{Clerk}] \sqcap \triangle[\texttt{salary}\colon (\epsilon < 4)] \sqcap \triangle[\texttt{boss}\colon \triangle[\texttt{salary}\colon (\epsilon > 6)]]\}$ $(\rightarrow_\sigma)$

$\mathbf{K} := \mathbf{K} \cup \{x_2/ \triangle[\texttt{boss}\colon \texttt{Clerk}], x_2/ \triangle[\texttt{salary}\colon (\epsilon < 4)], x_2/ \triangle[\texttt{boss}\colon \triangle[\texttt{salary}\colon (\epsilon > 6)]]\}$ $(\rightarrow_\sqcap)$

$\mathbf{K} := \mathbf{K} \cup \{x_2 \triangle x_4, x_4/[\texttt{salary}\colon (\epsilon > 6)], x_4/[\texttt{boss}\colon \texttt{Clerk}], x_4/[\texttt{salary}\colon (\epsilon < 4)],$
$x_4/[\texttt{boss}\colon \triangle[\texttt{salary}\colon (\epsilon > 6)]]\}$ $(\rightarrow_\triangle$ and $\rightarrow_{replace})$

No more rules are applicable to x_2. We may apply rules to x_4:

$\mathbf{K} := \mathbf{K} \cup \{x_4 \texttt{boss} x_5, x_5/\texttt{Clerk}, x_4 \texttt{salary} x_6, x_6/(\epsilon > 6), x_6/(\epsilon < 4), x_5/ \triangle[\texttt{salary}\colon (\epsilon > 6)]\}$ $(\rightarrow_{[]}$ and $\rightarrow_{replace})$

Again, no more rules are applicable to x_4. We may apply rules to successors of x_4:

$\mathbf{K} := \mathbf{K} \cup \{x_5/ \triangle[\texttt{boss}\colon \texttt{Clerk}] \sqcap \triangle[\texttt{salary}\colon (\epsilon < 4)] \sqcap \triangle[\texttt{boss}\colon \triangle[\texttt{salary}\colon (\epsilon > 6)]]\}$ $(\rightarrow_\sigma)$

$\mathbf{K} := \mathbf{K} \cup \{x_5/ \triangle[\texttt{boss}\colon \texttt{Clerk}], x_5/ \triangle[\texttt{salary}\colon (\epsilon < 4)], x_5/ \triangle[\texttt{boss}\colon \triangle[\texttt{salary}\colon (\epsilon > 6)]]\}$ $(\rightarrow_\sqcap)$

$\mathbf{K} := \mathbf{K}^{x_2}_{x_5} = \{x_4 \texttt{boss} x_2, \dots\}$ $(\rightarrow_{replace}$ due to $x_5 \equiv_K x_2)$

One can verify that no rule can be applied to $\mathbf{K}^{x_2}_{x_5}$; since it contains the base constraint $\{x_6/(\epsilon < 4), x_6/(\epsilon > 6)\}$, which is obviously unsatisfiable, the class `Clerk` is inconsistent.

4.3 Extending paths with multi–valued attributes

As mentioned in subsection 2.2, a path is undefined on set values. If the expression of a multi-valued attribute in a path is permitted, we need an extension of the semantics of paths of subsection 2.2 and, consequently, a change in the interpretation of atomic predicates of subsection 2.4.

Let us show first the problem by an example, exploiting **some** and **all** (see the examples in [36] related to query languages in an OODBMS environment) to scan over a set.

$$\begin{array}{rcl} \underline{\textbf{class}}\ \texttt{DangerWarehouse} & = & \underline{\textbf{isa}}\ \texttt{Warehouse}\ \underline{\textbf{and}}\ (\texttt{stock.risk}\ \underline{\textbf{some}} > 5) \\ \underline{\textbf{class}}\ \texttt{NoDangerWarehouse} & = & \underline{\textbf{isa}}\ \texttt{Warehouse}\ \underline{\textbf{and}}\ [\texttt{maxrisk: Integer}] \\ & & \underline{\textbf{and}}\ (\texttt{stock.risk}\ \underline{\textbf{all}} \leq \texttt{maxrisk}) \end{array}$$

For a given element of `DangerWarehouse`, the path `stock.risk` identifies a set of values, i.e. the risks of all the materials which are stocked there. The integrity constraint (`stock.risk` **some** > 5) is satisfied if at least one stocked material has a risk greater than 5. The integrity constraint (`stock.risk` **all** $\leq$ `maxrisk`) is satisfied if each stocked material has a risk less or equal than the one specified by the attribute `maxrisk`. As a matter of fact, the comparison between a path including multi–valued attributes and a base value (as in the case of (`stock.risk` **some** > 5)) can be easily expressed in $\mathcal{OLCP}$:

$$\sigma_P(\texttt{DangerWarehouse}) \quad = \quad \texttt{Warehouse} \sqcap\ \triangle[\texttt{stock: } \{(\triangle.\texttt{risk} > 5)\}_{\exists}]$$

On the other hand, we cannot express the comparison between two paths if one or both include multi–valued attributes (as in the case of (`stock.risk` **all** $\leq$ `maxrisk`)). For this purpose, we introduce the symbol $\exists$ as an element of a path to represent set membership.

Formally, a *path* p is a dot-separated sequence of elements $e_1.e_2.\cdots.e_n$, where $e_i \in \mathbf{A} \cup \{\triangle, \exists\}$. For example, the classes `DangerWarehouse` and `NoDangerWarehouse` are mapped below into the formalism:

$$\begin{array}{rcl} \sigma_P(\texttt{DangerWarehouse}) & = & \texttt{Warehouse} \sqcap\ (\triangle.\texttt{stock}.\exists.\ \triangle\ .\texttt{risk} > 5) \\ \sigma_P(\texttt{NoDangerWarehouse}) & = & \texttt{Warehouse} \sqcap\ \triangle[\texttt{maxrisk: Integer}] \\ & & \sqcap\neg(\triangle.\texttt{stock}.\exists.\ \triangle\ .\texttt{risk} > \triangle.\texttt{maxrisk}) \end{array}$$

The semantics of $\exists$ is given by:

$$\mathcal{J}[\exists] \quad = \quad \left\{(v', v'') \in \mathcal{V} \times \mathcal{V} \mid v'' \in v'\right\}$$

Let v be a value and p be a path. By $\mathcal{J}[p](v)$ we mean here the set of values reachable from v following p, that is $\{v' \mid (v, v') \in \mathcal{J}[p]\}$. Consequently, we change the interpretation of atomic predicates as follows:

$$\begin{array}{rcl} \mathcal{I}[(p\theta d)] & = & \{v \in \mathcal{V} \mid \mathcal{J}[p](v) \cap \{d' \mid d'\theta d\} \neq \emptyset\} \\ \mathcal{I}[(p_1\theta p_2)] & = & \{v \in \mathcal{V} \mid (\mathcal{J}[p_1](v) \times \mathcal{J}[p_2](v)) \cap \{(d_1, d_2) \mid d_1\theta d_2\} \neq \emptyset\} \\ \mathcal{I}[(p\uparrow)] & = & \{v \in \mathcal{V} \mid \mathcal{J}[p](v) = \emptyset\} \end{array}$$

With these hypotheses we prove in Appendix B the following theorem:

Theorem 4.5 (Paths and Undecidability) *Checking consistency in a $\mathcal{OLCP}$ schema with multi-valued path components is an undecidable problem.*

5 Related Work and Discussion

In the literature we find works related to constraint satisfiability in the areas of deductive databases, conceptual modelling and description logics.

As a general comment, our approach differs from the perspective taken in deductive database theory [21, 38, 41, 42]: the latter relies on highly general inference mechanisms, usually facing undecidable problems, operating on a logical, flat model, while we propose a relatively simple and complete consistency checking procedure operating on types populated by complex values, including objects, relations, tuple and set constructors. Thus,

we limit in some way the type constructors and constraint expressiveness in order to achieve the decidability of the consistency problem; nevertheless, the formalism is powerful enough to express integrity rules, which play an outstanding role in the expression of integrity constraints. On the other hand, the extension of a flat model, such as the multi-sorted logic of [38], to support structured values is technically quite complex, as has been shown in Section 3. In addition, it is usually recognized that a general theorem prover running on a restricted language should be less efficient than a reasoner which is specialized on that restricted language.

Paper [38] is a forerunner in using the tableaux calculus technique to check constraints satisfiability in logic databases. Its static constraints are first order multi-typed logic sentences, restricted to finite worlds to achieve decidability.

The works related to the SATCHMO project [21, 41, 42] deal with the problem of constraints satisfiability in a deductive database environment. They consider a greatly extended set of constraints, namely closed first order formulas. This set has a semi-decidable unsatisfiability problem, however allows an inference procedure which is claimed to be satisfactorily efficient. The solution proposed in [41] follows a model-generation paradigm, since a set of constraints is satisfiable if a model, possibly empty, can be built for it; additional controls are included to find regular patterns that could give rise to infinite models. Unlike these works, we consider to be contradictory a constraint set which either does not allow any model or which forces some classes to be empty, while the algorithms in [41] should be slightly extended to individuate empty models.

F-Logic [37] extends a deductive environment towards the object-oriented paradigm. It is a *full-fledged* logic with a model-theoretic semantics and a sound and complete proof theory. Its type constructors include union, intersection, complement, universal and existential quantifiers. F-logic, being a general theorem prover and having some capabilities for issuing queries on the database schema, can be programmed to compute the satisfiability of a type. Anyway, F-logic does not express the comparison operators, which we use in the path expressions, and does not deal directly with paths and complex values, even if the authors claim they can be easily added as a syntactic sugar.

On the conceptual modelling side, we find many works which deal with different languages and propose ways to represent different forms of integrity constraints [8, 22, 23, 26, 51, 50]. As a general comment, most of them neither subsume nor are subsumed by our representation language. It is hard to say which is the best set of modeling features to be considered, but our language is the only one coupling powerful logical operators (intersection, union and complement) and complex structures (class types distinct from value types, sets and tuples).

In [26, 50, 51] the notion of declarative integrity constraints is introduced and the inference problem (i.e., whether a constraint is implied by a given set of constraints) is solved. In [51] path functional dependencies for semantic data models were first introduced and studied; in [50] path functional dependencies and path equations (i.e. path relations limitated to equality operator) for a data model supporting complex object types (no inheritance) have been considered in combination; in [26] a unified theory of typing constraint, path equations, and functional dependencies in Object Oriented Data models is proposed. The constraints they can declare are different from ours, since they do not include union and complement, but can express functional dependencies. They present a complete axiomatization, provide decidability results and, in particular, they provide a decision procedure for the inference problem for acyclic schemata. The inference problem is different from the consistency problem, but one can be reduced to the other, provided that complement is available.

Papers [22, 23] present extended description logics for database environment and provide consistency checking algorithms. Paper [23] presents a language which can be used to model various class-based representation formalisms, including object oriented languages, and proves that satisfiability checking is decidable. With respect to our formalism, the language on the one hand offers the inverse of the function represented by an atomic attribute, number restrictions and cyclic definitions, but, on the other, does not support value-types and path relations, which are essential for the representation of significant integrity constraints in object oriented environment. A similar comment can be made with respect to the work in [22] where the authors present a new data model which extends object oriented models in various directions, including n-ary relations, inverse of an attribute and cardinality constraints and provide schema satisfiability procedures.

In the introduction we recalled the works in the description logics area which inspired us. To our purpose,

description logics languages are not sufficient, since they do not consider concrete domains (which cannot be given up in database environment) and, for instance, the class `Technician` of section 1.1 could not be defined for instance in [28]. The only exception is found in [5] where concrete domains are integrated into concept languages ($\mathcal{ALC}(\mathcal{D})$). But this isn't still enough, because $\mathcal{ALC}(\mathcal{D})$ does not allow equality predicates on abstract domains and the class `BossTechnician` of section 1.1 could not be defined.

Let us now discuss the limits of our work. Our objective was mainly to deal with both classes and relations, and to represent inheritance, aggregation, if then rules, type constraints and path relations. Unfortunately path relations turned out to be in conflict with the cyclicity of the schema, and this lead us to propose two separate formalisms. Essentially our representation is inherently intensional: the adopted reasoning technique is not well suited for the aspects more related to instances, such as non-local aggregate constraints.

Inclusion dependency constraints are represented in a straightforward way when deriving from *isa* hierarchy and referential integrity, while general inclusion dependencies between unrelated classes (i.e. *multi-class constraints*) need some work, since we deal only with a single class at a time. On the other hand, thanks to the object oriented paradigm, this class of constraints could be represented by defining a new aggregate class where the appropriate constraints can be set on a single (aggregate) class basis.

Functional dependencies and *key constraints* are not represented, since our technique is centered on the intensional level. On the other hand, as functional dependencies do not imply negations, they cannot give rise to unsatisfiability, therefore they are essential when populating or querying a database but immaterial in our context.

Cardinality dependencies are represented only in the limited form of existential quantification. According to the results in [29] it is reasonable to suppose that the satisfialility problem in $\mathcal{OLCD}$ plus cardinalities is decidable with exponential space. However, dealing with cardinalities in object environment with complex values is technically very complex and this direction has yet to be explored. With respect to the complexity of the various possible flavours of our formalisms, the general results coming from description logics cannot be directly exploited in our context, mainly because of path expressions.

To conclude this discussion, we can observe that one major advantage of the tableaux calculus technique is its modularity: it can be extended by adding rules dealing with additional modelling features.

6 Conclusions

A database is usually ruled by a complex set of rules (known as integrity constraints) which limit the set of legal database states. As a practical solution, these integrity constraints can be enforced by various mechanisms, depending on the DBMS operating environment, but in general it is not possible to know in advance if there will be allowed non-empty database states. Our objective was to represent integrity constraints in a declarative style at the database schema level and find out which database classes cannot be filled by instances, because of the integrity constraints. Our focus was on the object oriented database environment, which is, in our opinion, most promising as the database paradigm in the near future. We imported the techniques of Description Logics and devised a theoretical framework for the description of an object schema with integrity constraints and a procedure for checking schema coherence. In particular, we discussed two formalisms (sharing a common kernel), whose union leads to undecidability. The formalisms can be the basis of a DDL able to represent classes, types and state integrity constraints. The checking procedure is a specialized reasoner based on the tableaux calculus and ensures completeness and termination. The computational properties of the reasoning algorithms are also given.

We implemented a modular database design tool based on the ancestor of formalisms presented here [6, 11, 14]. The tool takes as input as schema and a type description and checks if the type is consistent. The $\mathcal{OLCP}$ and $\mathcal{OLCD}$ reasoners are being integrated in this tool.

The usefulness of the techniques we propose is not limited to satisfiability: as we can perceive queries as virtual classes, the coherence procedure can be exploited to compute subsumption relationships between a query and the classes of a given schema. Our technique could then be used for the semantic query optimization. Subsumption computation for query optimization has been shown in [9, 19] for a quite simple DDL environment,

in [50] for the object oriented environment with path equations and functional dependencies and, for the DDL of the the present work, in [6, 11, 13, 14]. Further investigations are planned on this topic.

A Appendix

In this section, we give proofs of the logical properties of the calculi for $\mathcal{OLCP}$ and $\mathcal{OLCD}$, namely termination, soundness and completeness.

We assume a schema in normal form, denoted by σ, and types occurring in σ, denoted by S, possibly with subscripts or accents. We further assume that every constraint system $\mathbf{K}$ derives from an initial constraint system of the form $\{\hat{x}/\hat{S}\}$. We write $\mathbf{K} \to \mathbf{K}'$ iff $\mathbf{K}'$ is derived from $\mathbf{K}$ by an application of a completion rule. With the symbol $\vdash$ we denote the transitive closure of $\to$. By *subtype of* S we mean a substring of S that is a type.

We define the nested successor relation among variables as follows.

Definition A.1 *Let x, y be variables, $\mathbf{K}$ a constraint system. We say that x is a* nested successor *of y ($x \succ y$) in $\mathbf{K}$ iff there exists $e \in \mathbf{A} \cup \{\exists, \forall\}$ such that $yex \in \mathbf{K}$.*

Lemma A.2 *Let $\mathbf{K}$ contain no clash. Then $\succ$ is well-founded.*

PROOF SKETCH: Suppose $\succ$ is not well-founded. Then an infinite descending sequence $x_0 \prec x_1 \prec x_2 \prec \cdots$ exists. Theorem A.7 implies there are only finitely many variables in $\mathbf{K}$. Therefore, there exist distinct i, j such that $x_i = x_j$. By definition A.1, $\mathbf{K}$ includes a set $\{x_i e_i x_{i+1},\ x_{i+1} e_{i+1} x_{i+2},\ \ldots,\ x_{j-1} e_{j-1} x_j\}$, with $e_k \neq \triangle$ for $i \leq k < j$. This is a clash in $\mathbf{K}$ (clash 12).

□

A.1 $\mathcal{OLCP}$

Let $\mathbf{N}_\sigma$ denote the set $\mathbf{N} \setminus \mathbf{B} \cup \{\bot, \top\}$ of *schema names*. A *schema subtype of* S is either a subtype of S, or a subtype of the description of a type name that is a schema subtype of S. To simplify definitions, we assume each normal form type S in σ is associated to a *formation tree*, which is a tree representation of the prefix form of S, where $\sqcap, \sqcup$ are binary operators, $\neg, a, \triangle, \exists, \forall$ are unary operators, and a type name in $\mathbf{N}_\sigma$ is a unary operator whose argument is $\sigma(N)$, and each type name in $\mathbf{B} \cup \{\bot, \top\}$ or path equation $p\theta p'$, $p\theta d$, $p{\uparrow}$ is an atom. Obviously, every formation tree is finite, since every type description is finite and the schema is acyclic. Let S' be a schema subtype of S. By *abstract path to S' in S* we mean the nonempty sequence of operators which dominate the point representing S' in the formation tree of S.

Definition A.3 *The* size $|S|$ *of a type S is defined as:*

1. *1, if $S \in \mathbf{B} \cup \{\bot, \top\}$;*
2. *$|\sigma(S)| + 1$, if $S \in \mathbf{N}_\sigma$;*
3. *the number of symbols in S that are not type names plus the size of all type names in S, otherwise.*

Definition A.4 *Let S use S'. The* set level $\ell(S')$ of S' in S *is the number of $\exists$, $\forall$ symbols in the abstract path to S' in S.*

Lemma A.5 *Let $\{\hat{x}/\hat{S}\} \vdash \mathbf{K}$ and $y/S \in \mathbf{K}$, $y/S' \in \mathbf{K}$. Then S, S' have equal set level in $\hat{S}$.*

PROOF SKETCH: Let $y \simeq y'$ if and only if $yey' \in \mathbf{K}$, with $e \in \mathbf{A} \cup \{\triangle\}$, or $ye{\epsilon}y' \in \mathbf{K}$, or $y = y' \in \mathbf{K}$, and let $\sim$ denote the reflexive, symmetric and transitive closure of $\simeq$. We prove that if $y/S \in \mathbf{K}$, $y'/S' \in \mathbf{K}$, $y \sim y'$, then S, S' have equal set level in $\hat{S}$. The proof proceeds by induction: We assume $\mathbf{K} \to \mathbf{K}'$ and the claim holds for $\mathbf{K}$ and prove it holds for $\mathbf{K}'$ by cases of the $\to$ relation.

□

To prove termination, we use a result due to Manna and Dershowitz [40].

Theorem A.6 (Manna and Dershowitz, 1979) *Let $(X, \succ)$ be a well-founded ordering, and let $\mathbf{X}$ be the set of all multisets over X. Let $\gg$ be an ordering on $\mathbf{X}$ obtained by extending the ordering $\succ$ as follows: If $\mathcal{M}, \mathcal{M}'$ are multisets over X, then $\mathcal{M} \gg \mathcal{M}'$ iff $\mathcal{M}'$ can be obtained from $\mathcal{M}$ by replacing one or more elements by a finite number of elements, each of which is smaller than one of the replaced elements. Then $\gg$ is well-founded.*

Theorem A.7 (Termination) *Let $\hat{S}$ be a type, $\hat{x}$ a variable. There is no infinite sequence of constraint systems $\mathbf{K}_1 = \{\hat{x}/\hat{S}\}$, $\mathbf{K}_2$, ... , such that $\mathbf{K}_i \to \mathbf{K}_{i+1}$, for $i \geq 1$.*

PROOF SKETCH: Let $\mathbf{N}_\sigma(\hat{S})$ denote the subset of $\mathbf{N}_\sigma$ of names which are schema subtypes of $\hat{S}$. We say that a constraint κ is *typed* if its form is x/S. We say that κ is *complex* if and only if it is typed and $S \notin \mathbf{B} \cup \{\bot, \top\}$. We say that κ is *functional* if and only if its form is $xe.py$. We associate to every functional or typed κ in $\mathbf{K}$ an integer number $p(\kappa, \mathbf{K})$ which decreases when a completion rule having the constraint among its premises is applied. Let $n_\exists(\ell)$ be the total number of schema subtypes of $\hat{S}$ of the form $\{S\}_\exists$ having the same set level ℓ, and let

$$n_\exists = \max_\ell \{n_\exists(\ell)\}. \tag{22}$$

We define:

$$p(\kappa, \mathbf{K}) = \begin{cases} |S|(n_\exists + 1) + |\mathbf{N}_\sigma(\hat{S})| - c(\kappa, \mathbf{K}), & \text{if } \kappa \text{ is complex} \\ |e.p|, & \text{if } \kappa \text{ is functional} \\ 0, & \text{otherwise.} \end{cases} \tag{23}$$

where $c(\kappa, \mathbf{K})$ is the number of rule instances whose premises and conclusions are subsets of $\mathbf{K}$, such that κ is one of the premisses. Let $\tau_0(\mathbf{K})$ be a multiset of all -1's, one for every variable occurring in $\mathbf{K}$. Let

$$\tau(\mathbf{K}) = \{p(\kappa, \mathbf{K}) : \kappa \in \mathbf{K}\} \cup \tau_0(\mathbf{K}) \tag{24}$$

The main steps of the proof are the following. We prove by case analysis that if κ is functional or complex, then $c(\kappa, \mathbf{K}) \leq n_\exists + 1 + |\mathbf{N}_\sigma(\hat{S})|$. Therefore, since $|S| \geq 1$, $p(\kappa, \mathbf{K}) \geq 0$. Hence $n \in \tau(\mathbf{K})$ implies $n \geq -1$. Since $c(\kappa, \mathbf{K}') > c(\kappa, \mathbf{K})$, if κ is a premise of a rule instance $\mathbf{K} \to \mathbf{K}'$, then $p(\kappa, \mathbf{K}) > p(\kappa, \mathbf{K}')$. If κ, κ' are functional or typed, and κ is a premise and κ' a consequence of a rule instance $\mathbf{K} \to \mathbf{K}'$, then $p(\kappa, \mathbf{K}) > p(\kappa', \mathbf{K}')$. Finally, if $\mathbf{K} \to \mathbf{K}'$ is an instance of a rule, then $\tau(\mathbf{K}) \gg \tau(\mathbf{K}')$. In fact, the $\to_{replace}$-rule deletes from $\tau(\mathbf{K})$ an occurrence of -1. If other rules are applied and κ a premise and κ' a consequence of the rule instance, then $\mathbf{K}'$ is obtained from $\mathbf{K}$ by

1. deleting $p(\kappa, \mathbf{K})$, if κ' is not typed or functional;
2. otherwise replacing $p(\kappa, \mathbf{K})$ by $p(\kappa, \mathbf{K}')$, $p(\kappa', \mathbf{K}')$, -1, which are smaller than $p(\kappa, \mathbf{K})$

We have thus defined a termination function

$$\tau \colon \mathcal{K} \to \mathcal{M}(\mathcal{N} \cup \{-1\})$$

and proved that

$$\tau(\mathbf{K}) \gg \tau(\mathbf{K}') \quad \text{if} \quad \mathbf{K} \to \mathbf{K}'. \tag{25}$$

This property implies termination: An infinite sequence of constraint systems $\mathbf{K}_1 = \{\hat{x}/\hat{S}\}$, $\mathbf{K}_2$, ... , such that $\mathbf{K}_i \to \mathbf{K}_{i+1}$, for $i \geq 1$, is an instance of a completion rule, yields an infinite descending sequence of multisets, contradicting Theorem A.6.

□

In the following, we prove soundness and completeness of the completion rules. We assume a type $\hat{S}$ in σ and an initial variable $\hat{x}$; every constraint system $\mathbf{K}$ is derived by $\{\hat{x}/\hat{S}\}$ using completion rules. We define now a successor relation among constraints.

Definition A.8 *Let $\mathbf{K}$ be a constraint system, κ, $\kappa' \in \mathbf{K}$ constraints. κ is a successor of κ' ($\kappa \dashv\!\vdash \kappa'$) in $\mathbf{K}$ iff there exists a rule instance whose premise includes κ' and whose conclusion includes κ.*

Lemma A.9 *The successor relation among constraints is well founded.*

PROOF SKETCH: Suppose an infinite sequence $\kappa_0 \Vdash \kappa_1 \Vdash \kappa_2 \Vdash \cdots$ exists. Theorem A.7 implies $\kappa_i = \kappa_{i'}$ for some i, i'. As every constraint either has no successor, or is shorter than its predecessor, this contradicts $\kappa_i = \kappa_{i'}$. □

To prove soundness and completeness, we need the two following lemmata.

Lemma A.10 *A complete constraint system containing no clash is satisfiable.*

PROOF SKETCH: Let $\mathbf{K}$ be a clash-free complete constraint system derived from $\{\hat{x}/\hat{S}\}$. Note that $x/\bot$, $x \neq x$ are not in $\mathbf{K}$ by hypothesis (see clash 3, 4), and $x/\top$, $x \epsilon x$, and $x = x$ are satisfied by any $\mathcal{I}$-assignment α. These constraints are thus ignored in the following. Let $\bar{\mathcal{O}}$, $\mathbf{A}$ be disjoint denumerable sets, $\mathcal{X}$ be the set of variable symbols occurring in $\mathbf{K}$. Let also $\omega\colon \mathcal{X} \to \bar{\mathcal{O}}$, $\beta\colon \mathcal{X} \to \{a \in \mathbf{A} : a \text{ not a substring of any } \kappa \in \mathbf{K}\}$ be injective functions. Proceed inductively on $\succ$ to define α. Note that we write $x \neq y$ for both $y \neq x$ and $x \neq y$. **Base:** If a base constraint system $\mathbf{K}_b \subseteq \mathbf{K}$ exists, then an $\mathcal{I}$-assignment α_b exists satisfying $\mathbf{K}_b$, since its unsatisfiability is a clash (clash 1). A stronger statement holds, namely that an $\alpha_b \models \mathbf{K}_b$ can be found such that if $x = \{d_0, \ldots, d_m\} \in \mathbf{K}$, and y occurs in $\mathbf{K}_b$, and $x \exists y$, or $x \forall y$, is in $\mathbf{K}$, then $\alpha_b(y) = d_i$ for some $i \in 0..m$. Such an α_b exists because:

1. clashes 1, 14 and 15 do not hold by hypothesis;
2. y never occurs in constraints of the form $x' \theta x''$, because such constraints derive only from types of the form $p' \theta p''$, where p', p'' do not include sets.

Let $x_0, x_1, \ldots, x_n$ be the variables with no nested successors. For $i \in 0..n$, we define $\alpha(x_i)$ as follows. If x_i occurs in $\mathbf{K}_b$, $\alpha(x_i) = \alpha_b(x_i)$. Else if x_i is the left variable of a constraint $x_i \triangle y$, let $\alpha(x_i) = \omega(x_i)$. Else if $x_i = \emptyset \in \mathbf{K}$, define $\alpha(x_i) = \emptyset$. Otherwise, let $\alpha(x_i) = [\beta(x_i)\colon 0]$. **Step:** Let $y_1, \ldots, y_n$ be the nested successors of x. Notice that x does not occur in $\mathbf{K}_b$, because x has successors and $\mathbf{K}$ contains no clash (see clash 2); thus α is undefined on x. We have the following cases:

1. every e_i is an attribute a_i, and $i \neq j$ implies $e_i \neq e_j$, $i, j \in 1..n$.
2. every e_i is an existential symbol $\exists$, except at most one e_i which is a universal symbol $\forall$.

In fact, both $\{x \exists y_i, x a y_j\}$ and $\{x \forall y_i, x a y_j\}$ are clashes. Moreover, if $i \neq j$ then $x a y_i$, $x a y_j$ are not both in $\mathbf{K}$, because $\mathbf{K}$ is complete, hence it is complete w.r.t. $\rightarrow_{replace}$. In case 1 let $\alpha(x) := [a_1\colon \alpha(y_1), \ldots, a_n\colon \alpha(y_n), \beta(x)\colon 0]$. In case 2, if $x = \{d_1, \ldots, d_m\} \in \mathbf{K}$, let $\alpha(x) = \{d_1, \ldots, d_m\}$. Otherwise proceed as follows. Let $\mathbf{Z}_<$ ($\mathbf{Z}_>$) be the intersection of all sets $R(x, \mathbf{K})$ that contain a left (right) semi-interval minus the union of all finite sets $R(x, \mathbf{K})$ and the image of α_b, and let $\gamma_<\colon \mathcal{X} \to \mathbf{Z}_<$ ($\gamma_>\colon \mathcal{X} \to \mathbf{Z}_>$) be an injective function. If $x \forall y_i \in \mathbf{K}$, for some $i \in 1..n$, then let v_x be defined as $t \cup \{\beta(x)\colon 0\}$ if $\alpha(y_i)$ is a tuple t, $\omega(x)$ if $\alpha(y_i)$ is an object identifier, $\gamma_>(x)$ if $\alpha(y_i)$ is a base value and $R(y_i, \mathbf{K})$ contains a right semi-interval, $\gamma_<(x)$ if $\alpha(y_i)$ is a base value and $R(y_i, \mathbf{K})$ does not contain a right semi-interval. Finally, let $\alpha(x) := \{\alpha(y_1), \ldots, \alpha(y_n), v_x\}$ if $x \forall y_i \in \mathbf{K}$, for some $i \in 1..n$; $\alpha(x) := \{\alpha(y_1), \ldots, \alpha(y_n)\} \cup \{\beta(x) : 0\}$ otherwise. We can now build δ. Let $\delta(\alpha(x)) := \alpha(y)$ if $x \triangle y \in \mathbf{K}$, and $\delta(\omega(x)) = \delta(\alpha(y))$ if $x \forall y \in \mathbf{K}$ and $v_x = \omega(x)$.

Finally, we build $\mathcal{I}$, first for primitive names, then for virtual names. For any primitive name N, let $\mathcal{I}[N] := \{\alpha(x) \mid x/N \in \mathbf{K}\}$. For any virtual name N, let $\mathcal{I}[N] := \mathcal{I}[\sigma(N)]$. Let $\mathbf{K}$ be a constraint system derived from $\{\hat{x}/\hat{S}\}$. For $\kappa \in \mathbf{K}$, let $\Omega(\kappa)$ be defined by:

1. $\Omega(\hat{x}/\hat{S}) = 0$
2. $\Omega(\kappa) = n$ where n is the maximum m such that $\kappa \dashv\!\vdash \kappa_{m-1} \dashv\!\vdash \cdots \dashv\!\vdash \kappa_1 \dashv\!\vdash \hat{x}/\hat{S}$

We prove by induction on Ω that $\alpha \models \mathbf{K}$, in the form: $\alpha \models \kappa \Leftarrow$ for all κ' with $\Omega(\kappa') > \Omega(\kappa)\colon \alpha \models \kappa'$.

$\alpha \models x/(\epsilon = d)$, $x/(\epsilon < d)$, $x/(\epsilon > d)$, $x > y$, $x < y$, x/B. Every such constraint is in $\mathbf{K}_b$ and $\alpha \models \mathbf{K}_b$.

$\alpha \models x/\neg B$. If $x/\neg B \in \mathbf{K}_b$, it follows from $\alpha \models \mathbf{K}_b$. Otherwise by construction every $\alpha(x)$ is a complex value.

$\alpha \models x \neq y$. Assume $\alpha(x) = \alpha(y)$. If $\alpha(x)$ is a tuple t, it contains a pair $\beta(x) : 0$. By the domain and injectivity of β, $\beta(x) : 0$ is not in $\alpha(y)$. If $\alpha(x)$ is an object identifier, it differs from $\alpha(y)$ since ω is injective. If $\alpha(x)$ is a set, then proceed by case analysis. Assume first $x = \{d_1, \ldots, d_m\} \in \mathbf{K}$. Then, $y = \{d_1, \ldots, d_m\} \in \mathbf{K}$ cannot hold (clash 13), and in the remaining cases, $\alpha(y)$ contains either a non-base value, or a base value not within the bounds of $\{d_1, \ldots, d_m\}$. Assume now $x = \{d_1, \ldots, d_m\} \notin \mathbf{K}$. Notice in this case that $\beta(x) : 0$ and v_x are not in $\alpha(y)$ (by the domain and injectivity of $\omega, \beta, \gamma_>, \gamma_<$)

$\alpha \models x = \emptyset$. Since $\mathbf{K}$ contains no clash, x has no successors (see clash 10). Therefore by construction, $\alpha(x)$ is a base value, an object identifier, or $\emptyset$, but none of the first two cases may hold because $\mathbf{K}$ contains no clash (see clash 11, 10).

$\alpha \models x/a\uparrow$ By construction, if $\alpha(x) = [a_1 \colon v_1, \ldots, v_p]$, then for some $i \in 1..p$ $xa_iy_i \in \mathbf{K}$ or $a_i : v_i$ is the pair $\beta(x) = 0$. In both cases, a_i differs from a (by clash 7 and the definition of β).

$\alpha \models x/\triangle\uparrow$. $\alpha(x) = o$ implies $x \triangle y \in \mathbf{K}$, which clashes with $x/\triangle\uparrow$ (see clash 7).

$\alpha \models x/\neg\{\top\}_\forall$. Assume $\alpha(x)$ is a set. Then $x = \emptyset \in \mathbf{K}$, or $\{d_1, \ldots, d_p\} \in \mathbf{K}$ for some $d_1, \ldots, d_p$, or $x\exists y \in \mathbf{K}$, or $x\forall y \in \mathbf{K}$ for some x, y. By the structure of completion rules, either case holds only if $x/\{S'\}_\forall \in \mathbf{K}$, for some S', which clashes with x/S (clash 9).

$\alpha \models xay, x \triangle y$. By construction.

$\alpha \models x\exists y, x\forall y$. By construction $\alpha(x)$ is a set. If $x = \{d_1, \ldots, d_m\} \notin \mathbf{K}$, then $\alpha(y) \in \alpha(x)$ by construction. Otherwise $\alpha(y) \in \alpha(x)$ holds by the property of α_b.

$\alpha \models x/\neg N$, N primitive. Suppose $\alpha(x) \in \mathcal{I}[N]$. Then, by construction, y exists such that $y/N \in \mathbf{K}$ and $\alpha(y) = \alpha(x)$. But x and y are not the same variable, because $\{x/\neg N, x/N\}$ is a clash (see clash 5); therefore, by the completeness hypothesis $x \neq y \in \mathbf{K}$, and by induction, $\alpha \models x \neq y$. Hence $\alpha(y) \neq \alpha(x)$. Contradiction.

$\alpha \models xe.py$, $x/p\theta p'$, $x/p\theta d$; $x/\{S\}_\exists$, $x/\triangle S$, $x/[a \colon S]$; $x/S \sqcup S'$, $x/S \sqcap S'$. By construction and the completeness hypothesis.

$\alpha \models x/\{S\}_\forall$. If $x = \{d_1, \ldots, d_m\} \in \mathbf{K}$, we know that $\{d_1, \ldots, d_m\} \subseteq R(y, \mathbf{K})$ and it can be shown that $R(y, \mathbf{K}) \subseteq \mathcal{I}[S]$, (for a y such that $x\forall y \in \mathbf{K}$). Otherwise, by the completeness hypothesis, y_i/S (for $1 \leq i \leq n$) are in $\mathbf{K}$. Then $\alpha(y_i) \in \mathcal{I}[S]$ by the inductive hypothesis. v_x is in $\mathcal{I}[S]$ since:

- if t is in $\mathcal{I}[S]$ then also $t \cup \{\beta(x) \colon 0\}$ is;
- if $v_x = \omega(x)$ then $\delta(v_x) = \delta(\omega(x)) = \delta(\alpha(y))$
- if v_x is a base value, notice that $R(y, \mathbf{K})$ is infinite, thus it contains a right semi-interval or a left semi-interval.

$\alpha \models x/\neg N$, where N is virtual; x/N. These can be proven using induction, the completeness hypothesis on $\mathbf{K}$, and the construction. It can also be proven that $\alpha(x) \in \mathcal{I}[N]$ implies $\alpha(x) \in \mathcal{I}[\sigma(N)]$ if N is primitive, namely that $\mathcal{I}$ is indeed a legal instance.

□

Lemma A.11 *Let $\mathbf{K}$ be satisfiable, incomplete constraint system. Then there is a satisfiable constraint system $\mathbf{K}'$ such that $\mathbf{K} \to \mathbf{K}'$.*

PROOF SKETCH: We show by cases how to build an $\mathcal{I}$-assignment α' satisfying $\mathbf{K}'$ from an $\mathcal{I}$-assignment α satisfying $\mathbf{K}$. Let X be the set of variables occurring in $\mathbf{K}$. Let α be an $\mathcal{I}$-assignment such that $\alpha \models \mathbf{K}$. We consider relevant cases only.

$\to_\sigma$. The $\mathcal{I}$–assignment α satisfies $x/N \in \mathbf{K}$, that is $\alpha(x) \in \mathcal{I}[N]$. But $\mathcal{I}$ is a legal instance, therefore $\mathcal{I}[N] \subseteq \mathcal{I}[\sigma(N)]$. Hence α satisfies $\{x/\sigma(N)\} \cup \mathbf{K}$.

$\to_\neg$. The $\mathcal{I}$–assignment α satisfies $x/\neg N \in \mathbf{K}$, that is $\alpha(x) \in \mathcal{I}[\neg N] = \mathcal{V}(\mathcal{O}) \setminus \mathcal{I}[N]$. But $\mathcal{I}$ is a legal instance, therefore $\mathcal{I}[N] = \mathcal{I}[\sigma_V(N)]$, whence $\alpha(x) \in \mathcal{V}(\mathcal{O}) \setminus \mathcal{I}[\sigma_V(N)] = \mathcal{I}[\neg\sigma_V(N)]$. Hence α satisfies $\{x/\neg\sigma_V(N)\} \cup \mathbf{K}$. □

$\to_{\exists\forall}$. We have $\{x/\{S\}_\forall, x\exists y\} \subseteq \mathbf{K}$ and $\mathbf{K}' = \{y/S\} \cup \mathbf{K}$. As $\alpha(x) \subseteq \mathcal{I}[S]$, and α satisfies $x\exists y$, $\alpha(y) \in \alpha(x)$, we have $\alpha(y) \in \mathcal{I}[S]$.

$\to_\exists$. $x/\{S\}_\exists \in \mathbf{K}$ and $\mathbf{K}' = \{x\exists y, y/S\} \cup \mathbf{K}$. As α satisfies $x/\{S\}_\exists$, we have $\alpha(x) = \{\ldots, v, \ldots\}$, with $v \in \mathcal{I}[S]$. As α is undefined on y (y is a new variable), define an extension α' to $X \cup \{y\}$ of the assignment α by $\alpha'(y) := v$ and $\mathcal{I}' := \mathcal{I}$.

$\to_{[]}$. $x/[a\colon S] \in \mathbf{K}$ and $\mathbf{K}' = \{xay, y/S\} \cup \mathbf{K}$. Since α satisfies $\mathbf{K}$, $\alpha(x)$ is a tuple $[\dots, a\colon v, \dots]$, with $v \in \mathcal{I}[S]$. As α is undefined on y (y is a new variable), define an extension α' to $X \cup \{y\}$ of the assignment α by $\alpha'(y) := v$ and $\mathcal{I}' := \mathcal{I}$.

$\to_{\triangle}$. We have $x/ \triangle S \in \mathbf{K}$ and $\mathbf{K}' = \{x \triangle y, y/S\} \cup \mathbf{K}$. As α satisfies $\mathbf{K}$, $\alpha(x) = o$, with $\delta(o) \in \mathcal{I}[S]$. Since y is new, define an extension α' to $X \cup \{y\}$ of the assignment α by $\alpha'(y) = \delta(o)$.

$\to_{\theta}$. We have $x/p\theta p' \in \mathbf{K}$ and $\mathbf{K}' = \{xpy, xp'z, y\theta z\} \cup \mathbf{K}$. $\alpha \models \mathbf{K}$, hence $\mathcal{J}[p](\alpha(x))\theta\mathcal{J}[p'](\alpha(x))$ holds. As y, z are new, define $\alpha(y) := \mathcal{J}[p](\alpha(x))$ and $\alpha(z) := \mathcal{J}[p'](\alpha(x))$.

$\to_{path}$. We have $x/e.py \in \mathbf{K}$ and $\mathbf{K}' = \{xez, zpy\} \cup \mathbf{K}$. $\alpha \models \mathbf{K}$, hence $\alpha(y) = \mathcal{J}[e] \circ \mathcal{J}[p](\alpha(x))$. Since z is new, and $\mathcal{J}[e]$ is defined on $\alpha(x)$, define $\alpha(z) := \mathcal{J}[e](\alpha(x))$.

The completeness theorem follows from the above lemmata.

Theorem A.12 (Completeness) *S is consistent if and only if some completion $\mathbf{K}$ of $\{x/S\}$ contains no clash.*

PROOF SKETCH: Recall that S is consistent if and only if $\{x/S\}$ is satisfiable.

$\Longrightarrow$. Assume S is consistent. Then by Lemma A.11 some completion $\mathbf{K}$ of $\{x/S\}$ is satisfiable. A satisfiable constraint system contains no clash, since a clash is unsatisfiable.

$\Longleftarrow$. Since $\mathbf{K}$ is a clash-free completion of $\{x/S\}$, it is satisfiable by Lemma A.10; as $\mathbf{K} \supseteq \{x/S\}$, S is consistent. □

A.2 $\mathcal{OLCD}$

In the following, we prove soundness and completeness of the completion rules. We assume a type $\hat{S}$ in σ and an initial variable $\hat{x}$; every constraint system $\mathbf{K}$ is derived by $\{\hat{x}/\hat{S}\}$ using completion rules.

We define now a successor relation among constraints.

Definition A.13 *Let $\mathbf{K}$ be a constraint system, $\kappa, \kappa' \in \mathbf{K}$ constraints. κ is a successor of κ' ($\kappa \dashv \kappa'$) in $\mathbf{K}$ if and only if κ is of the form x/S, κ' is of the form x'/S', and S is a subtype of S'.*

Lemma A.14 *The successor relation among constraints is well-founded.*

PROOF SKETCH: Since every subtype of a type S is shorter than S, there cannot be an infinite sequence $\kappa_0 \vdash \kappa_1 \vdash \kappa_2 \vdash \cdots$.

□

To prove soundness and completeness, we need the two following lemmata.

Lemma A.15 *A complete constraint system containing no clash is satisfiable.*

PROOF SKETCH: Let $\bar{\mathcal{O}}$, $\mathbf{A}$ be disjoint denumerable sets, $\mathcal{X}$ be the set of variable symbols occurring in $\mathbf{K}$. Let also $\omega\colon \mathcal{X} \to \bar{\mathcal{O}}$, $\beta\colon \mathcal{X} \to \{a \in \mathbf{A} : a \text{ not a substring of any } \kappa \in \mathbf{K}\}$ be injective functions, and α_b an assignment satisfying $\mathbf{K}_b$ such that if $x = \{d_0, \dots, d_m\} \in \mathbf{K}$, and y occurs in $\mathbf{K}_b$, and $x\exists y$, or $x\forall y$, is in $\mathbf{K}$, then $\alpha_b(y) = d_i$ for some $i \in 0..m$. Let $\alpha\colon \mathcal{X} \to \mathcal{V}$ be a map satisfying the following equations:

$$\alpha(x) = \begin{cases} \alpha_b(x) & \text{if } x \text{ occurs in } \mathbf{K}_b, \\ \emptyset & \text{if } x = \emptyset \in \mathbf{K}, \\ \omega(x) & \text{if } x \triangle y \in \mathbf{K}, \\ [a_1\colon \alpha(y_1), \dots, a_n\colon \alpha(y_n), \beta(x) : 0] & \text{if } xa_iy_i \in \mathbf{K} \text{ for } i \in 1 \dots n, \\ \{d_1, \dots, d_m\} & \text{if } x = \{d_1, \dots, d_m\} \in \mathbf{K}, \\ \{\alpha(y_i) : 1 \le i < n\} \cup \{\alpha(y), v_x\} & \text{if } x = \{d_1, \dots, d_m\} \notin \mathbf{K} \text{ and } x\exists y_i \in \mathbf{K} \text{ for } 1 \le i < n \text{ and } x\forall y \in \mathbf{K}, \\ \{\alpha(y_i) : 1 \le i \le n\} \cup \{[\beta(x) : 0]\} & \text{if } x = \{d_1, \dots, d_m\} \notin \mathbf{K} \text{ and } x\exists y_i \in \mathbf{K} \text{ for } 1 \le i \le n, \\ [\beta(x) : 0] & \text{otherwise,} \end{cases}$$

where n is the number of nested successors of x and

$$v_x = \begin{cases} t \cup \{\beta(x) : 0\} & \text{if } \alpha(y) \text{ is a tuple } t, \\ \omega(x) & \text{if } \alpha(y) \text{ is an object identifier,} \\ \gamma_>(x) & \text{if } \alpha(y) \text{ is a base value and } R(y, \mathbf{K}) \text{ contains a right semi-interval,} \\ \gamma_<(x) & \text{if } \alpha(y) \text{ is a base value and } R(y, \mathbf{K}) \text{ does not contain a right semi-interval.} \end{cases}$$

where $\gamma_< : \mathcal{X} \to \mathbf{Z}_<$ ($\gamma_> : \mathcal{X} \to \mathbf{Z}_>$) is an injective function, $\mathbf{Z}_<$ ($\mathbf{Z}_>$) is the intersection of all sets $R(x, \mathbf{K})$ that contain a left (right) semi-interval minus the union of all finite sets $R(x, \mathbf{K})$ and the image of α_b. Clash-freeness implies α is defined by recursion on $\succ$. Finally, let $\delta(\omega(x))$ be $\alpha(y)$ if $x \triangle y \in \mathbf{K}$, $\delta(\alpha(y))$ if $\omega(x) = v_x$, and any value otherwise; let $\mathcal{I}[N] = \{\alpha(x) \mid x/N \in \mathbf{K}\}$ if N is a primitive type name, and $\mathcal{I}[N] = \{\alpha(x) \mid x/N \in \mathbf{K}$ or $x/\sigma(N) \in \mathbf{K}\}$ if N is a virtual type name.
$\alpha \models \mathbf{K}$ can be proved by induction on $\dashv$. We skip routine cases.

Base. $\alpha \models x \neq y$. Assume $\alpha(x) = \alpha(y)$. If $\alpha(x)$ is a tuple t, it contains a pair $\beta(x) : 0$. By the domain and injectivity of β, $\beta(x) : 0$ is not in $\alpha(y)$. If $\alpha(x)$ is an object identifier, it differs from $\alpha(y)$ since ω is injective. If $\alpha(x)$ is a set, then proceed by case analysis. Assume first $x = \{d_1, \ldots, d_m\} \in \mathbf{K}$. Then, $y = \{d_1, \ldots, d_m\} \in \mathbf{K}$ cannot hold (clash 13), and in the remaining cases, $\alpha(y)$ contains either a non-base value, or a base value not within the bounds of $\{d_1, \ldots, d_m\}$. Assume now $x = \{d_1, \ldots, d_m\} \notin \mathbf{K}$. Notice in this case that $\beta(x) : 0$ and v_x are not in $\alpha(y)$ (by the domain and injectivity of $\omega, \beta, \gamma_>, \gamma_<$).

$\alpha \models x/\neg\{\top\}_\forall$. Assume $\alpha(x)$ is a set. Then $x = \emptyset \in \mathbf{K}$, or $\{d_1, \ldots, d_p\} \in \mathbf{K}$ for some $d_1, \ldots, d_p$, or $x\exists y \in \mathbf{K}$, or $x\forall y \in \mathbf{K}$ for some x, y. By the structure of completion rules, either case holds only if $x/\{S'\}_\forall \in \mathbf{K}$, for some S', which clashes with x/S (clash 9).

$\alpha \models x\exists y, x\forall y$. By construction $\alpha(x)$ is a set. If $x = \{d_1, \ldots, d_m\} \notin \mathbf{K}$, then $\alpha(y) \in \alpha(x)$ by construction. Otherwise $\alpha(y) \in \alpha(x)$ holds by the property of α_b.

$\alpha \models x/N$, $x/\neg N$. The former is satisfied by definition. Assume $\alpha(x) \in \mathcal{I}[N]$. Then, x/N is in $\mathbf{K}$, or $x/\sigma(N)$ is in $\mathbf{K}$. Since $\mathbf{K}$ is complete, $x/\sigma(N)$ is surely in $\mathbf{K}$. Hence there is a clash in $\mathbf{K}$ (see clash 6). Contradiction.

Step. $\alpha \models x/\{S\}_\forall$. If $x = \{d_1, \ldots, d_m\} \in \mathbf{K}$, we know that $\{d_1, \ldots, d_m\} \subseteq R(y, \mathbf{K})$ and it can be shown that $R(y, \mathbf{K}) \subseteq \mathcal{I}[S]$ (for a y such that $x\forall y \in \mathbf{K}$). Otherwise, by the completeness hypothesis, y_i/S (for $1 \leq i \leq n$) are in $\mathbf{K}$. Then $\alpha(y_i) \in \mathcal{I}[S]$ by the inductive hypothesis. v_x is in $\mathcal{I}[S]$ since:

- if t is in $\mathcal{I}[S]$ then also $t \cup \{\beta(x) : 0\}$ is;
- if $v_x = \omega(x)$ then $\delta(v_x) = \delta(\omega(x)) = \delta(\alpha(y))$
- if v_x is a base value, notice that $R(y, \mathbf{K})$ is infinite, thus it contains a right semi-interval or a left semi-interval.

□

Lemma A.16 *Let* $\mathbf{K}$ *be a satisfiable incomplete constraint system. Then there is a satisfiable constraint system* $\mathbf{K}'$ *such that* $\mathbf{K} \to \mathbf{K}'$.

PROOF SKETCH: For all rules, the arguments used for the proof sketch of Theorem A.11 work, with the exception of the $\to_{replace}$-rule of page 19, case 2b, where K–equivalence must be used. □

The completeness theorem follows from the above lemmata.

Theorem A.17 (Completeness) *S is consistent if and only if some completion* $\mathbf{K}$ *of* $\{x/S\}$ *contains no clash.*

PROOF SKETCH: Same as Theorem A.12. □

B Undecidability results

Let Σ be an alphabet of symbols. By Σ^* we denote, as usual, the set of all finite sequences (words) of symbols from Σ, including the empty sequence ϵ. The set Σ^* is a semigroup with respect to the associative operation of concatenation of words (the *free semigroup on* Σ). Let R be a finite binary relation

$$R = \{(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)\}$$

of elements (u_i, v_i) in Σ^*, and define a binary relation $\rightarrow_R$ on Σ^* by:

$$u \rightarrow_R v \quad \Leftrightarrow \quad \exists w_1, w_2 \in \Sigma^* \; \exists (u_i, v_i) \in R : \; u = w_1 u_i w_2 \wedge v = w_1 v_i w_2$$

The symbol $\tilde{R}$ is used to denote the equivalence relation generated by $\rightarrow_R$. The binary relation $\tilde{R}$ determines a congruence on Σ^*, and the quotient set $\Sigma^*/\tilde{R}$ is itself a semigroup whose elements are the equivalence classes of $\tilde{R}$. If Σ is finite, a semigroup isomorphic to $\Sigma^*/\tilde{R}$ is said to have a *finite presentation* $\langle \Sigma \mid R \rangle$. The *word problem for semigroups* is the problem to decide $u\tilde{R}v$ for given words $u, v \in \Sigma^*$.

The problem was shown undecidable in [44]. Gurevich has later strengthened the result, extending it to finite semigroups [33]. In the sequel we assume that a word $w = w_1 w_2 \ldots w_n$ in Σ^* is written as a dot-separated sequence of symbols $w_1 . w_2 . \ldots . w_n$, to avoid clumsy notation for the path associated to w.

We prove the undecidability of checking consistency both of a cyclic $\mathcal{OLCP}$ schema with descriptive semantics and of a $\mathcal{OLCP}$ schema with multi-valued path components, by reduction of the word problem in finite semigroups.

B.1 Proof of theorem 4.1

The reduction uses a construction originally in [43].

Proposition B.1 *Let A be a finite alphabet and $\Sigma = \{\triangle . a \colon a \in A\}$. Let $\langle \Sigma \mid R \rangle$ be a finite presentation of a finite semigroup and let σ be the following schema:*

$$\sigma(N) = \triangle(\bigsqcap_{a \in A}[a \colon N]) \sqcap \bigsqcap_{i \leq n}(u_i = v_i)$$

Then $u\tilde{R}v$ if and only if $N \sqcap \neg(u = v)$ is inconsistent, for any $u, v \in \Sigma^*$.

Proof.

($\Rightarrow$). Let $\mathcal{I}$ be an instance of σ. As $\rightarrow_R$ generates the equivalence $\tilde{R}$, if $u\tilde{R}v$, then either $u = v$, or there is w such that $u\tilde{R}w$ and either $w \rightarrow_R v$, or $v \rightarrow_R w$. Assume $u \rightarrow_R v$, where $u = xu_iy$ and $v = xv_iy$. We know that for all $o \in \mathcal{I}[N]$ and for all $x \in \Sigma^*$: $\mathcal{J}[x](o) \subseteq \mathcal{I}[N]$ (because of $\triangle\bigsqcap_{a \in A}[a \colon N]$ in $\sigma(N)$). It follows that $\mathcal{J}[xu_i](o) = \mathcal{J}[xv_i](o)$ (because of $\bigsqcap_{i \leq n}(u_i = v_i)$ in $\sigma(N)$), hence $\mathcal{J}[u](o) = \mathcal{J}[v](o)$. By induction, we may conclude that $u\tilde{R}v$ implies $\mathcal{J}[u](o) = \mathcal{J}[v](o)$, for all $o \in \mathcal{I}[N]$. Thus $\mathcal{I}[N] \subseteq \mathcal{I}[(u = v)]$ whence $N \sqcap \neg(u = v)$ is inconsistent if $u\tilde{R}v$.

($\Leftarrow$). We assume that $\mathcal{I}[N \sqcap \neg(u = v)] = \emptyset$ for all instances $\mathcal{I}$ of σ. Let $x\tilde{R}$ denote the equivalence class of x w.r.t. to $\tilde{R}$. Now we construct a particular instance of σ as follows:

$$\mathcal{O} = \{x\tilde{R} \mid x \in \Sigma^*\}$$

$$\delta(x\tilde{R}) = [\ldots, a \colon (x . \triangle . a)\tilde{R}, \ldots] \qquad \text{for all } a \in A$$

$$\mathcal{I}[N] = \mathcal{O}$$

$\mathcal{I}$ is an instance of σ since

1. for all $o \in \mathcal{I}[N]$ it holds that $\mathcal{J}[a](o) \in \mathcal{I}[N]$ for all $a \in A$;

2. for all o, $\mathcal{J}[u_i](o) = \mathcal{J}[v_i](o)$ for all pairs (u_i, v_i) of R. In fact, $\mathcal{J}[\triangle.a](x\tilde{R}) = \{(x.\triangle.a)\tilde{R}\}$ and, by induction, $\mathcal{J}[w](x\tilde{R}) = \{(x.w)\tilde{R}\}$. For all pairs (u_i, v_i) of R, we have that $u_i \tilde{R} v_i$, hence $(x.u_i)\tilde{R} = (x.v_i)\tilde{R}$, finally $\mathcal{J}[u_i](x\tilde{R}) = \mathcal{J}[v_i](x\tilde{R})$.
3. by hypothesis, $\{x\tilde{R} \mid x \in \Sigma^*\}$ is finite, and so is $\mathcal{O}$.

Because of our assumption ($\mathcal{I}[N \sqcap \neg(u = v)] = \emptyset$ and hence $\mathcal{I}[N] \subseteq \mathcal{I}[(u = v)]$), we know that $\mathcal{J}[u](o) = \mathcal{J}[v](o)$ for all $o \in \mathcal{I}[N] = \mathcal{O}$. Thus, in particular, we have that $\mathcal{J}[u](\epsilon\tilde{R}) = \mathcal{J}[v](\epsilon\tilde{R})$, hence $(\epsilon u)\tilde{R} = (\epsilon v)\tilde{R}$, and finally $u\tilde{R} = v\tilde{R}$, which means $u\tilde{R}v$.

□

B.2 Proof of theorem 4.5

The construction takes hint from the one used in [34], to prove an undecidability result for a terminological language.

Proposition B.2 *Let A be a finite alphabet, $\Sigma = \{\triangle.a \colon a \in A\}$, and let $\langle \Sigma \mid R \rangle$ be a finite presentation of a finite semigroup. Let* left, right, back, forth *be attributes and let:*

$$\mathsf{relcond} = u_1 = v_1 \sqcap u_2 = v_2 \sqcap \ldots \sqcap u_n = v_n \tag{26}$$

$$\mathsf{eq}_{u,v} = (\triangle.\mathsf{left} = \triangle.\mathsf{start}.u) \sqcap (\triangle.\mathsf{right} = \triangle.\mathsf{start}.v) \tag{27}$$
$$\sqcap (\triangle.\mathsf{forth}.\exists) = (\triangle.\mathsf{start})$$

$$\mathsf{loop}_a = \triangle[\mathsf{forth}\colon \{\triangle[a\colon \triangle(\mathsf{back}.\triangle.\mathsf{forth}.\exists{=}\epsilon)]\}_\forall] \tag{28}$$
$$\sqcap \neg(\triangle.\mathsf{forth}.\exists.\triangle.a.\triangle.\mathsf{back} \neq \epsilon)$$

where $1 \leq i \leq n$, $a \in A$, and $u, v \in \Sigma^*$.

Then the type

$$\mathsf{eq}_{u,v} \sqcap \bigsqcap_{a \in A} \mathsf{loop}_a \sqcap \triangle[\mathsf{forth}\colon \{\mathsf{relcond}\}_\forall]$$

is subsumed by $\triangle.\mathsf{left} = \triangle.\mathsf{right}$ if and only if $u\tilde{R}v$.

Proof.

($\Leftarrow$). Let S be the subsumee and let $\bar{o} \in \mathcal{I}[S]$. For every $o \in \mathcal{J}[\triangle.\mathsf{forth}.\exists](\bar{o})$ and $w \in \Sigma^*$ it can be proved that

$$\mathcal{J}[w](o) \subseteq \mathcal{J}[\triangle.\mathsf{forth}.\exists](\bar{o}) \tag{29}$$

(by induction on the length of w, using (28) in the base case). We prove that from $u\tilde{R}v$ follows $\mathcal{J}[\triangle.\mathsf{start}.u](\bar{o}) = \mathcal{J}[\triangle.\mathsf{start}.v](\bar{o})$. Assume first $u \to_R v$. By the definition of $\to_R$ we have $u = xu_iy$, $v = xv_iy$ and $(u_i, v_i) \in R$. Let o be the unique element in $\mathcal{J}[\triangle.\mathsf{start}](\bar{o})$. By (27), $o \in \mathcal{J}[\triangle.\mathsf{forth}.\exists](\bar{o})$. From (29) follows that $\mathcal{J}[x](o) \subseteq \mathcal{J}[\triangle.\mathsf{forth}.\exists](\bar{o})$, and thus, by the definition of S, $\mathcal{J}[x](o) \subseteq \mathcal{I}[\mathsf{relcond}]$. By (26) we have $\mathcal{J}[xu_iy](o) = \mathcal{J}[xv_iy](o)$, whence $\mathcal{J}[\triangle.\mathsf{start}.u](\bar{o}) = \mathcal{J}[\triangle.\mathsf{start}.v](\bar{o})$. The claim follows easily by induction. Finally by (27) $\mathcal{J}[\triangle.\mathsf{left}](\bar{o}) = \mathcal{J}[\triangle.\mathsf{right}](\bar{o})$.

($\Rightarrow$). Assuming $u\tilde{R}v$ does not hold, a legal instance can be found such that S is not subsumed by $\triangle.\mathsf{left} = \triangle.\mathsf{right}$. Let $\Sigma^*/\tilde{R}$ be the (finite) set of equivalence classes defined by $\tilde{R}$ and let $\mathcal{O} = \Sigma^*/\tilde{R} \cup \{\bar{o}\}$. The

legal instance is defined by:

$$
\begin{aligned}
\mathcal{J}[\triangle.\mathsf{start}](\bar{o}) &= \epsilon\tilde{R} \\
\mathcal{J}[\triangle.\mathsf{left}](\bar{o}) &= u\tilde{R} \\
\mathcal{J}[\triangle.\mathsf{right}](\bar{o}) &= v\tilde{R} \\
\mathcal{J}[\triangle.\mathsf{forth}.\exists](\bar{o}) &= \{x\tilde{R} |\, x \in \Sigma^*/\tilde{R}\} \\
\mathcal{J}[\triangle.\mathsf{back}](x\tilde{R}) &= \bar{o} && \text{for all } x \in \Sigma^*/\tilde{R} \\
\mathcal{J}[\triangle.a](x\tilde{R}) &= (x.\triangle.a)\tilde{R} && \text{for all } x \in \Sigma^*/\tilde{R} \text{ and } a \in A
\end{aligned}
$$

□

References

[1] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Trans. on Database Systems*, 12(4):525–565, 1987.

[2] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *SIGMOD*, pages 159–173. ACM Press, 1989.

[3] P. Atzeni, editor. *LOGIDATA$^+$: Deductive Databases with Complex Objects*, volume 701 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg - Germany, 1993.

[4] P. Atzeni and D. S. Parker. Formal properties of net-based knowledge representation schemes. *Data and Knowledge Engineering*, 3:137–147, 1988.

[5] F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. In *12th International Joint Conference on Artificial Intelligence.*, Sydney, Australia, 1991.

[6] J. P. Ballerini, D. Beneventano, S. Bergamaschi, C. Sartori, and M. Vincini. A semantics driven query optimizer for OODBs. In A. Borgida, M. Lenzerini, D. Nardi, and B. Nebel, editors, *DL95 - Intern. Workshop on Description Logics*, volume 07.95 of *Dip. di Informatica e Sistemistica - Univ. di Roma "La Sapienza" - Rapp. Tecnico*, pages 59–62, Roma, June 1995.

[7] F. Bancilhon, C. Delobel, and P. Kanellakis (editors). *Building an Object-Oriented Database Systems: The story of O2*. Morgan Kaufmann, San Mateo, CA, 1992.

[8] G. Di Battista and M. Lenzerini. Deductive entity relationship modeling. *IEEE Trans. on Knowledge and Data Engineering*, 5(3):439–450, June 1993.

[9] H. W. Beck, S. K. Gala, and S. B. Navathe. Classification as a query processing technique in the CANDIDE data model. In *5th Int. Conf. on Data Engineering*, pages 572–581, Los Angeles, CA, 1989.

[10] D. Beneventano and S. Bergamaschi. Subsumption for complex object data models. In J. Biskup and R. Hull, editors, *4th Int. Conf. on Database Theory*, pages 357–375, Heidelberg, Germany, October 1992. Springer-Verlag.

[11] D. Beneventano, S. Bergamaschi, S. Lodi, and C. Sartori. Using subsumption in semantic query optimization. In A. Napoli, editor, *IJCAI Workshop on Object-Based Representation Systems*, pages 19–31, August 1993.

[12] D. Beneventano, S. Bergamaschi, and C. Sartori. Taxonomic reasoning with cycles in LOGIDATA$^+$. In P. Atzeni, editor, *LOGIDATA$^+$: Deductive Databases with Complex Objects*, volume 701 of *Lecture Notes in Computer Science*, pages 105–128. Springer-Verlag, Heidelberg - Germany, 1993.

[13] D. Beneventano, S. Bergamaschi, and C. Sartori. Using subsumption for semantic query optimization in OODB. In *Int. Workshop on Description Logics*, volume D-94-10 of *DFKI - Document*, pages 97–100, Bonn, Germany, June 1994.

[14] Domenico Beneventano, Sonia Bergamaschi, and Claudio Sartori. Semantic query optimization by subsumption in OODB. In H. Christiansen, H. L. Larsen, and T. Andreasen, editors, *Flexible Query Answering Systems*, volume 62 of *Datalogiske Skrifter - ISSN 0109-9799*, Roskilde, Denmark, 1996.

[15] S. Bergamaschi and B. Nebel. The complexity of multiple inheritance in complex object data models. In *Workshop on AI and Objects - IJCAI '91*, Sidney - Australia, August 1991.

[16] S. Bergamaschi and B. Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Journal of Applied Intelligence*, 4:185–203, 1994.

[17] S. Bergamaschi and C. Sartori. On taxonomic reasoning in conceptual design. *ACM Trans. on Database Systems*, 17(3):385–422, September 1992.

[18] E. Bertino and L. Martino. *Object Oriented Database Systems*. Addison-Wesley, 1993.

[19] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick. CLASSIC: A structural data model for objects. In *SIGMOD*, pages 58–67, Portland, Oregon, 1989.

[20] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[21] F. Bry, H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In H. W. Schmidt, S. Ceri, and M. Missikoff, editors, *EDBT '88 - Advances in Database Technology*, pages 488–505, Heidelberg, Germany, March 1988. Springer-Verlag.

[22] D. Calvanese and M. Lenzerini. Making object-oriented schemas more expressive. In *PODS - Principles of Database Systems*. ACM Press, 1994.

[23] D. Calvanese, M. Lenzerini, and D. Nardi. A unified framework for class-based formalisms. In J. Allen, R. Fikes, and E. Sandewall, editors, *KR '94 - Int. Conf on Principles of Knowledge Representation and Reasoning*, pages 109–120, Cambridge - MA, 1994. Morgan Kaufmann Publishers, Inc.

[24] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984.

[25] R. G. G. Cattell, editor. *The Object Database Standard: ODMG93*. Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[26] N. Coburn and G. E. Weddel. Path constraints for graph-based data models: Towards a unified theory of typing constraints, equations and functional dependencies. In *2nd Int. Conf. on Deductive and Object-Oriented Databases*, pages 312–331, Heidelberg, Germany, December 1991. Springer-Verlag.

[27] L. M. L. Delcambre and K. C. Davis. Automatic validation of object-oriented database structures. In *5th Int. Conf. on Data Engineering*, pages 2–9, Los Angeles, CA, 1989.

[28] F. M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages. In J. Allen, R. Fikes, and E. Sandewall, editors, *KR '91 - 2nd Int. Conf on Principles of Knowledge Representation and Reasoning*, pages 151–162, Cambridge - MA, April 1991. Morgan Kaufmann Publishers, Inc.

[29] F. M. Donini, A. Schaerf, and M. Buchheit. Decidable reasoning in terminological knowledge representation systems. In *13th International Joint Conference on Artificial Inteligence*, September 1993.

[30] Francesco M. Donini, Bernhard Hollunder, Maurizio Lenzerini, Alberto Marchetti Spaccamela, Daniele Nardi, and Werner Nutt. The complexity of existential quantification in concept languages. *Artificial Intelligence*, 53(2-3):309–327, February 1992.

[31] T. Finin and D. Silverman. Interactive classification as a knowledge acquisition tool. In L. Kershberg, editor, *Expert Database Systems*, pages 79–90. The Benjamin/Cummings Publishing Company, 1986.

[32] H. Gallaire and J. M. Nicholas. Logic and databases: An assessment. In S. Abiteboul and P. Kanellakis, editors, *ICDT 90*, pages 177–186, Heidelberg, Germany, December 1990. Springer-Verlag.

[33] Yuri Gurevich. The word problem for certain classes of semigroups. *Algebra and Logic*, 5:25–35, 1966.

[34] Philipp Hanschke. Specifying role interaction in concept languages. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Proc. of the Third Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 318–329. Morgan Kaufmann Publishers, 1992.

[35] B. Hollunder, W. Nutt, and M. Schmidt-Schauss. Subsumption algorithms for concept description languages. In *9th ECAI*, pages 348–353, Stockholm, Sweden, 1990.

[36] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *SIGMOD '92*, pages 393–402. ACM, June 1992.

[37] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[38] C. Kung. A tableaux approach for consistency checking. In *IFIP WG 8.1 Conf. on Theoretical and Formal Aspects of Information Systems*, Sitges, Spain, April 1985.

[39] C. Lecluse and P. Richard. Modelling complex structures in object-oriented databases. In *Symp. on Principles of Database Systems*, pages 362–369, Philadelphia, PA, 1989.

[40] Zohar Manna and Nachum Dershowitz. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, August 1979.

[41] R. Manthey. Satisfiability of integrity constraints: Reflections on a neglected problem. In *Int. Workshop on Foundations of Models and Languages for Data and Objects*, Aigen, Austria, September 1990.

[42] R. Manthey and F. Bry. SATCHMO: a theorem prover implemented in prolog. In *CADE 88 - 9th Conference on Automated Deduction*. Springer-Verlag - LNCS, 1988. ECRC Tech. Rep. KB-21, Nov. 87.

[43] B. Nebel. Terminological cycles: Semantics and computational properties. In J. F. Sowa, editor, *Principles of Semantic Networks*, chapter 11, pages 331–362. Morgan Kaufmann Publishers, Inc., San Mateo, Cal. USA, 1991.

[44] Emil L. Post. Recursive unsolvability of a problem of Thue. *Journal of Symbolic Logic*, 12:1–11, 1947.

[45] D. J. Rosenkrantz and H. B. Hunt. Processing conjunctive predicates and queries. In *Proc. Int'l. Conf. on Very Large Data Bases*, pages 64–72, Montreal, Canada, October 1980.

[46] M. Schmidt-Schauss and G. Smolka. Attributive concept descriptions with unions and complements. *Artificial Intelligence*, 48(1), 1991.

[47] R. M. Smullyan. *First-Order Logic*. Springer-Verlag, Berlin, Germany, 1986.

[48] J. Sowa, editor. *Principles of Semantic Networks*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1991.

[49] M. Stonebraker. *Object Relational DBMSs*. Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[50] M. F. van Bommel and G. E. Weddel. Reasoning about equations and functional dependencies on complex objects. *IEEE Transactions on Knowledge and Data Engineering*, 6(3):455–469, June 1994.

[51] Grant E. Weddel. Reasoning about functional dependencies generalized for semantic data models. *ACM Trans. on Database Systems*, 17(1):32–64, March 1992.